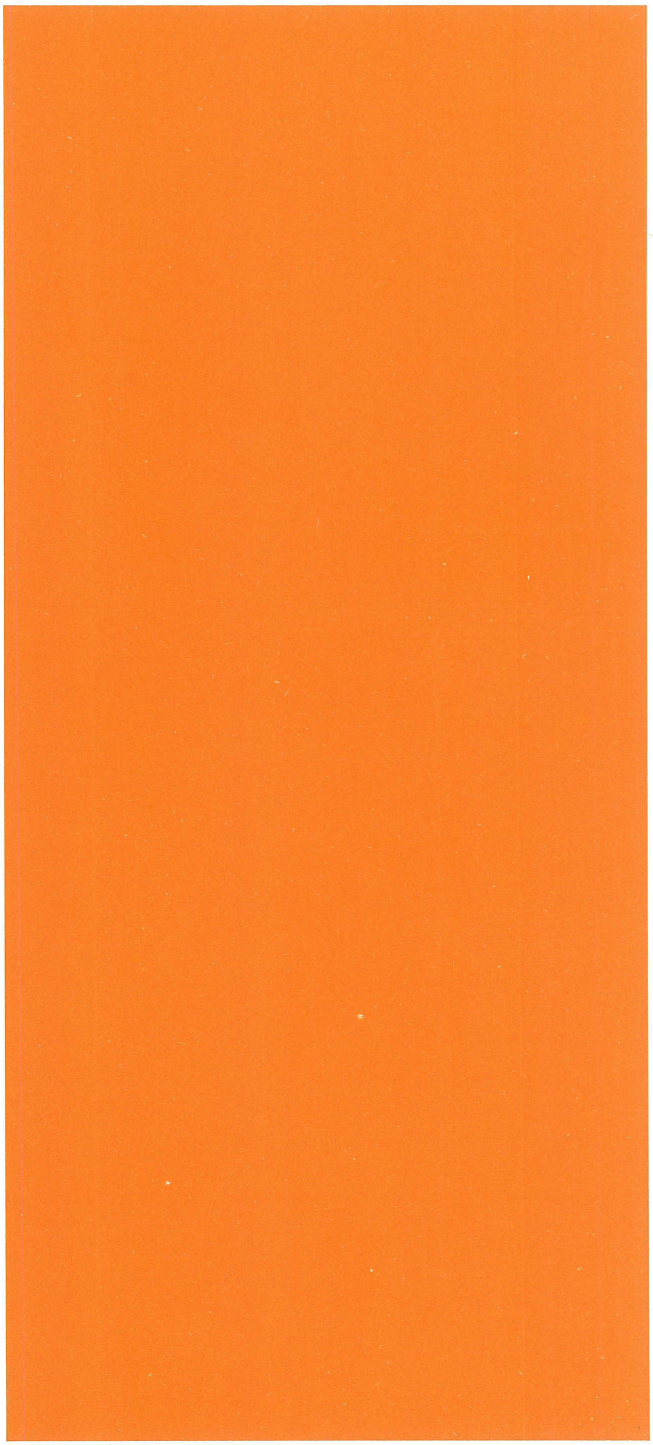
Honeywell Bull

JOVIAL

SERIES 600/6000

SOFTWARE



BS 06, Rev. 1

Ref.: 19.30.221 A

Honeywell Bull JOVIAL

SERIES 600/6000

SUBJECT:

Description and Use of the JOVIAL Programming Language in Both Batch and Time-Sharing Environments.

SPECIAL INSTRUCTIONS:

This manual supersedes the previous edition dated November 1971 and its Addendum A, dated June 1972. New and changed information is indicated by marginal change bars; asterisks indicate deletions.

SOFTWARE SUPPORTED:

Series 600 Software Release 7.0
Series 6000 Software Release E

DATE:

January 1973

ORDER NUMBER:

BS06, Rev. 1

Printed in France

Ref :19.30.221 A

PREFACE

This manual is intended as a reference for the programmer of the Honeywell Series 600/6000 JOVIAL language.

JOVIAL is a coded program designed to extend the power of Series 600/6000 in the area of program preparation and maintenance. It is supported by comprehensive documentation and training; periodic program maintenance and, where feasible, improvements are furnished for the current version of the program, provided it is not modified by the user.

© 1965, 1966, 1970, General Electric Company, U.S.A.
© 1971, 1972, 1973, Honeywell Information Systems, Inc.

File No.: 1623, 1723

FUNCTIONAL LISTING OF PUBLICATIONS
for
SERIES 600 SYSTEM

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 600:		
Hardware reference:			
Series 600	System Manual	371	BM78
DATANET 355	DATANET 355 Systems Manual	1645	BS03
Operating system:			
Basic Operating System	General Comprehensive Operating Supervisor (GCOS)	1518	BR43
Control Card Formats	Control Cards Reference Manual	1688	BS19
System initialization:			
GCOS Startup	System Operating Techniques	DA10	DA10
Communications System	GRTS/355 Startup Procedures Reference Manual	1715	BJ70
Storage Subsystem Startup	DSS180 Disk Storage Subsystem Startup Procedures	DA11	DA11
Data management:			
File System	File Management Supervisor	DB54	DB54
Integrated Data Store (I-D-S)	Integrated Data Store	1565	BR69
File Processing	Indexed Sequential Processor	DA37	DA37
Multi-Access I-D-S	Multi-Access I-D-S Implementation Guide	DA80	DA80
File Input/Output	General File and Record Control System	1003	BN85
I-D-S Data Query System	I-D-S Data Query System Installation	DB57	DB57
I-D-S Data Query System	I-D-S Data Query System User's Guide	DB56	DB56
Program maintenance:			
Object Program	Source and Object Library Editor	1723	BJ71
System Editing	System Library Editor	1687	BS18
Test system:			
On-Line Peripheral testing	GCOS On-Line Peripheral Test System (OPTS-600)	1573	BR76
Total On-Line testing	Total On-Line Test System (TOLTS)	DA49	DA49
Language processors:			
Macro Assembly Language	Macro Assembler Program	1004	BN86
COBOL Language	COBOL Compiler	1652	BS08
COBOL Usage	COBOL User's Guide	1653	BS09
ALGOL Language	ALGOL	1657	BS11
JOVIAL Language	JOVIAL	1650	BS06
FORTRAN Language	FORTRAN	1686	BJ67
FORTRAN IV Language	FORTRAN IV	1006	BN88
DATANET 355	DATANET 355 Macro-Assembler Program	1660	BB98
Generators:			
Sorting	Sort/Merge Program	1005	BN87
Merging	Sort/Merge Program	1005	BN87
Simulators:			
DATANET 355 Simulation	DATANET 355 Simulator Reference Manual	1663	BW23

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 600:		
Remote terminal system:			
DATANET 30	NPS/30 Programming Reference Manual	1558	BR68
DATANET 30/305/355	GRTS Programming Reference	DA79	DA79
Service and utility routines:			
Loader	General Loader	1008	BN90
Utility Programs	Utility	1422	BQ66
Conversion	Bulk Media Conversion	1096	BP30
System Accounting	GCOS Accounting Summary Edit Programs	1651	BS07
FORTRAN	FORTRAN Subroutine Libraries Reference Manual	1620	BR95
Controller Loader	Relocatable Loader	DA12	DA12
Service Routines	Service Routines	DA97	DA97
Software Debugging	Trace and Debug Routines	DB20	DB20
Time-sharing systems:			
Operating System	GCOS Time-Sharing System General Information	1643	BS01
System Programming	GCOS Time-Sharing Terminal/Batch Interface Facility	1642	BR99
System Programming	GCOS Time-Sharing System Programmers' Reference Manual	1514	BR39
BASIC Language	Time-Sharing BASIC	1510	BR36
FORTRAN Language	Time-Sharing FORTRAN	1566	BR70
Text Editing	Time-Sharing Text Editor	1515	BR40
Transaction processing:			
User's Procedures	Transaction Processing System User's Guide	DA82	DA82
Handbooks:			
Console Messages	Console Typewriter Messages	1477	BR09
Index	Comprehensive Index	1499	BR28
Pocket guides:			
Time-Sharing Programming	GCOS Time-Sharing System	1661	BS12
Macro Assembly Language	GMAP	1673	BS16
COBOL Language	COBOL	1689	BJ68
Control Card Formats	GCOS Control Cards & Abort Codes	1691	BJ69
Software maintenance (SMD):			
Table Definitions	GCOS Introduction & System Tables SMD	1488	BR17
Startup program	Startup (INIT) SMD	1489	BR18
Input System	System Input SMD	1490	BR19
Peripheral Allocation	Dispatcher and Peripheral Allocation SMD	1491	BR20
Core Allocation/Rollcall	Rollcall, Core Allocation and Operator Interface SMD	1492	BR21
Fault Processing	Fault Processing SMD	1493	BR22
Channel Modules	I/O Supervisor (IOS) SMD	1494	BR23
Error Processing	GCOS Exception Processing SMD	1495	BR24
Output System	Termination and System Output SMD	1496	BR25
File System Modules	File System Maintenance SMD	1497	BR26
Utility Programs	GCOS Utility Routines SMD	1498	BR27
Time-Sharing System	Time-Sharing Executive SMD	1501	BR29

Rev. 7303

FUNCTIONAL LISTING OF PUBLICATIONS
for
SERIES 6000 SYSTEM

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 6000:		
Hardware reference:			
Series 6000	Series 6000 Summary Description	DA48	DA48
DATANET 355	DATANET 355 Systems Manual	1645	BS03
Operating system:			
Basic Operating System	General Comprehensive Operating Supervisor (GCOS)	1518	BR43
Control Card Formats	Control Cards Reference Manual	1688	BS19
System initialization:			
GCOS Startup	System Startup and Operation	DA06	DA06
Communications System	GRTS/355 Startup Procedures Reference Manual	1715	BJ70
Storage Subsystem Startup	DSS180 Disk Storage Subsystem Startup Procedures	DA11	DA11
Data management:			
File System	File Management Supervisor	DB54	DB54
Integrated Data Store (I-D-S)	Integrated Data Store	1565	BR69
File Processing	Indexed Sequential Processor	DA37	DA37
Multi-Access I-D-S	Multi-Access I-D-S Implementation Guide	DA80	DA80
File Input/Output	General File and Record Control System	1003	BN85
I-D-S Data Query System	I-D-S Data Query System Installation	DB57	DB57
I-D-S Data Query System	I-D-S Data Query System User's Guide	DB56	DB56
Program maintenance:			
Object Program	Source and Object Library Editor	1723	BJ71
System Editing	System Library Editor	1687	BS18
Test system:			
On-Line Peripheral Testing	GCOS On-Line Peripheral Test System (OPTS-600)	1573	BR76
Total On-Line Testing	Total On-Line Test System (TOLTS)	DA49	DA49
Error Analysis and Logging	Honeywell Error Analysis and Logging System	DB50	DB50
Language processors:			
Macro Assembly Language	Macro Assembler Program	1004	BN86
COBOL Language	COBOL Compiler	1652	BS08
COBOL Usage	COBOL User's Guide	1653	BS09
ALGOL Language	ALGOL	1657	BS11
JOVIAL Language	JOVIAL	1650	BS06
FORTRAN Language	FORTRAN	1686	BJ67
DATANET 355	DATANET 355 Macro-Assembler Program	1660	BB98
Generators:			
Sorting	Sort/Merge Program	1005	BN87
Merging	Sort/Merge Program	1005	BN87

FUNCTION	APPLICABLE REFERENCE MANUAL TITLE	FORMER PUB. NO.	ORDER NO.
	Series 6000:		
Simulators:			
DATANET 355 Simulation	DATANET 355 Simulator Reference Manual	1663	BW23
Service and utility routines:			
Loader	General Loader	1008	BN90
Utility Programs	Utility	1422	BQ66
Conversion	Bulk Media Conversion	1096	BP30
System Accounting	GCOS Accounting Summary Edit Programs	1651	BS07
FORTRAN	FORTRAN Subroutine Libraries Reference Manual	1620	BR95
Controller Loader	Relocatable Loader	DA12	DA12
Service Routines	Service Routines	DA97	DA97
Software Debugging	Trace and Debug Routines	DB20	DB20
Time-sharing systems:			
Operating System	GCOS Time-Sharing System General Information	1643	BS01
System Programming	GCOS Time-Sharing Terminal/Batch Interface Facility	1642	BR99
System Programming	GCOS Time-Sharing System Programmers' Reference Manual	1514	BR39
BASIC Language	Time-Sharing BASIC	1510	BR36
FORTRAN Language	FORTRAN	1686	BJ67
Text Editing	Time-Sharing Text Editor	1515	BR40
Remote terminal system:			
DATANET 30	NPS/30 Programming Reference	1558	BR68
DATANET 30/305/355	GRTS Programming Reference	DA79	DA79
Transaction processing:			
User's Procedures	Transaction Processing System User's Guide	DA82	DA82
Handbooks:			
Console Messages	Console Typewriter Messages	1477	BR09
Index	Comprehensive Index	1499	BR28
Pocket guides:			
Time-Sharing Programming	GCOS Time-Sharing System	1661	BS12
Macro Assembly Language	GMAP	1673	BS16
COBOL Language	COBOL	1689	BJ68
Control Card Formats	GCOS Control Cards and Abort Codes	1691	BJ69

CONTENTS

		Page
Section I	Introduction.	1-1
	General.	1-1
	Series 600/6000 Computer Characteristics. . .	1-1
	Source Language Format	1-2
	Source Program Format.	1-2
	Program Listings	1-3
	Source Code Listing	1-3
	Object Code Listing	1-3
Section II	Signs and Constants	2-1
	Signs.	2-1
	Arithmetic Operators.	2-1
	Logical Operators	2-2
	Relational Operators.	2-2
	Separators.	2-3
	Enclosures.	2-3
	Type Descriptor	2-4
	Reserved JOVIAL Words	2-4
	Constants.	2-5
	Numeric	2-5
	Boolean	2-5
	Literal	2-5
	Status.	2-6
	Octal	2-6
Section III	Variables and Functions	3-1
	Variables.	3-1
	Special Variables.	3-2
	Functions.	3-4
Section IV	Simple Statements	4-1
	Statement Name	4-1
	Assignment Statement	4-1
	Exchange Statement	4-4
	GOTO Statement	4-4
	Switch Statement	4-4
	Stop Statement	4-5
	Compound Statement	4-6
Section V	Complex Statements.	5-1
	IF Statement	5-1
	IFEITH Statement	5-1
	FOR Statement.	5-2
	TEST Statement	5-4
Section VI	Data Declarations	6-1
	Item Descriptions.	6-1
	MODE Declaration	6-4
	Present Data List.	6-5
	DEFINE Declaration	6-5
	'PROGRAM Declaration	6-6

		Page
	TABLE Declaration.	6-6
	Ordinary Table Declaration.	6-7
	Specified Table Declaration	6-7
	Symbols Relating to Table Declarations. . . .	6-7
	Tables and Subordinate Overlays	6-9
	STRING Table Item Declaration	6-10
	LIKE Table Declaration.	6-11
	ARRAY Declaration.	6-12
	OVERLAY Declaration.	6-14
Section VII	Direct Code and Process Declarations.	7-1
	Direct Code.	7-1
	Direct Code Restrictions and Conventions	7-1
	Assign Statement.	7-3
	CLOSE Declaration.	7-5
	FUNCTION Declaration	7-6
	PROCEDURE Declaration.	7-7
	Interprocedure Communication.	7-10
	System Supplied Subprograms	7-11
	Closed Subroutines.	7-12
Section VIII	COMPOOL	8-1
	Communications Pool.	8-1
	COMPOOL Structure.	8-1
	Accessing of COMPOOL Definitions	8-2
	COMPOOL Assembly	8-3
	COMPOOL Disassembly.	8-6
	Sample Coding.	8-7
Section IX	Input and Output Operations	9-1
	FILE Operations.	9-1
	FILE Statement	9-1
	IOn Statement.	9-2
	IN Statement	9-4
	OUT Statement.	9-5
	WAIT Statement	9-6
Section X	Object Code Debug	10-1
	Debug Option	10-1
Section XI	Control Cards	11-1
	$ SNUMB.	11-1
	$ IDENT.	11-1
	$ OPTION	11-2
	$ JOVIAL	11-3
	$ EXECUTE.	11-4
	$ LIMITS	11-4
	$ ENDJOB	11-5
	***EOF	11-6
Appendix A	JOVIAL Compilation Error Messages	A-1
Appendix B	Machine Language (GMAP) Error Flags	B-1
Appendix C	Encoding of Signs	C-1
Appendix D	JOVIAL Language Restrictions.	D-1
Appendix E	JOVIAL Use Memos.	E-1

		Page
Appendix F	Time-Sharing JOVIAL	F-1
	Time-Sharing JOVIAL Subsystem.	F-1
	RUN Command.	F-2
	Usage.	F-5
	FILE Characteristics	F-7
	RUN Examples	F-8
	Accessing the Time-Sharing JOVIAL Subsystem.	F-10
	Terminal	F-10
	Paper Tape I/O in Time-Sharing JOVIAL.	F-12
	Restrictions.	F-12
	Time-Sharing Abort Messages	F-12
	Faults in Time-Sharing Execution	F-13
	Interpreting Error Diagnostics	F-13
	Time-Sharing JOVIAL Source Program Restrictions.	F-15
Appendix G	System Editor Interface	G-1
Appendix H	JOVIAL Compile Abort Codes.	H-1
Appendix I	Cross-Reference Table	I-1
Index		i-1

ILLUSTRATIONS

		Page
Figure 1-1	Syntax Example.	1-5
Figure 1-2	Usage Error Example	1-5
Figure 4-1	Valid Assign Statement Combination.	4-3
Figure 7-1	Sample JOVIAL Program with Mixed Subroutines.	7-13
Figure 8-1	Generating a COMPOOL Tape	8-8
Figure 8-2	Use of the COMPOOL Tape	8-9
Figure 9-1	Example of I/O Print Information.	9-6

SECTION I

INTRODUCTION

GENERAL

The JOVIAL (Jules Own Version International Algebraic Language) programming language consists, basically, of statements and declarations.

- Statements specify computations to be performed
- Declarations name and describe data on which the program is to operate

Statements and declarations, in turn, are composed of symbols -- words of the JOVIAL language. And symbols are composed of signs that constitute the JOVIAL alphabet.

Statements and declarations are translated by the JOVIAL compiler and the program is executed by the Comprehensive Operating Supervisor. Only those rules and symbols which govern JOVIAL programming on a Series 600/6000 computer system are considered in this manual.

Series 600/6000 Computer Characteristics

JOVIAL is not entirely a computer-independent programming language. Certain features of the language depend for their interpretation on characteristics of the computer for which the programs are to be written. Language restrictions for JOVIAL are listed in Appendix D.

The following characteristics of the computer system should be considered when programming in JOVIAL:

- Word size - 36 bits. The left-most bit is considered bit position 0; the right-most bit is considered bit position 35.
- Internal core storage - 36-bit, fixed length binary word, maximum 256k words, using an 18-bit address.
- Arithmetic - two's complement.

  a. Fixed point: 36-bit signed full word; 72-bit signed double word.

b. Floating point: 8-bit signed exponent, 28-bit signed mantissa.

c. Numbers are interpreted as binary coded decimal unless programmer specified.

- Character representation - 6-bit encoding.
- External storage - magnetic tape, drum, disk, punched cards, and printed material.

SOURCE LANGUAGE FORMAT

The source program is keypunched from an 80-column coding sheet, each line representing one card of JOVIAL source code. The JOVIAL statement may begin in any column from column 1-72 as free field format. Columns 73 through 80 are reserved for deck identification and sequencing, and are listed by the compiler. A $ must never appear in column 1 of a JOVIAL source program card.

Since the end of a JOVIAL statement is indicated by a $, it is possible for several statements to occupy the same punched card. If the last statement on the card was not completed prior to column 73, the statement may be continued on the next card. The compiler recognizes column 1 as following column 72 of the preceding card.

Blanks are used as separators; wherever one is permitted, any number are allowed.

Comments may be added to or embedded in any JOVIAL statement simply by enclosing the comment with a double apostrophe at the beginning and ending of the comment.

SOURCE PROGRAM FORMAT

The first card of the source program deck must be an identification card. The compiler uses this card to create a SYMDEF to be used by the loader. There are three types of source input to the compiler -- a program, a procedure, and data to generate a COMPOOL (COMmunications POOL). The format of the identification card for each of these is as follows:

Program

Columns 1 to 6 contain PROGRM. Columns 8 to 13 may be blank or may contain a name comprised of six characters or less, left-justified, to be used as the SYMDEF. If columns 8 to 13 are blank, a SYMDEF, NONAME, is created.

Procedure (External)

Columns 1 to 6 contain PROCED. Columns 8 to 13 may be blank or may contain a name comprised of six characters or less, left-justified, to be used as the SYMDEF. If columns 8 to 13 are blank, a SYMDEF, NONAME, is created.

Note

The utility of an unnamed procedure is questionable.

COMPOOL Generation

Columns 1 to 6 contain GENCOM. Columns 8 to 13 may be blank or may contain a name comprised of six characters or less, left-justified, to be used as the SYMDEF. If columns 8 to 13 are blank, a SYMDEF, NONAME, is created.

Immediately following the identification card is the START card, the first statement in every JOVIAL program. START is not followed by a $. JOVIAL statements which make up the program then follow. The last statement in every JOVIAL program must be

```
TERM $
```

or

```
TERM _ statement name $
```

The statement name, if used, will be that of the first statement in the program to be executed.

The effect of the TERM statement is to generate a subprogram epilogue, restoring index registers and returning control to the calling routine. In the case of a main program, control is returned to the operating system for wrapup and termination of execution.

PROGRAM LISTINGS

Source Code Listing

Source language statements are printed out after each compilation. The printout is representative of the punched card image and has a one line per one punched card format. There is, in addition to the JOVIAL statement information on each line, a corresponding number appended to each margin. The number in the left margin is the statement number of the last statement begun on that line. The number in the right margin represents a sequence number for each punched card. (See Figure 1-1.)

Syntactical (construction or grammatic) errors are identified at the time of the infraction or as soon thereafter as they are detected and an error message is printed. The error message appears in the stream of source code. Semantic (usage) error messages are printed immediately following the source listing and are associated with the statement in error by the identifying statement number. Appendix A lists compilation error messages.

Object Code Listing

The LSTOU option must be selected on the $ JOVIAL control card in order to have an object code listing produced after a compilation. If a JOVIAL statement has not been compiled because of a generated error or because of code not acceptable, the object code will be flagged with an alpha character on the left margin.

Figure 1-1 illustrates an example of a TABLE declaration without a corresponding END statement. The compiler discovers the error upon encountering the symbol DEFINE in statement 6. The arrow indicates the point of discovery, as does the error character count 000000. In this particular occurrence, there is no affect on the compilation, since the compiler assumes that the TABLE declaration is finished (which it is).

Figure 1-2 illustrates a sample of JOVIAL source code, semantic error messages, and machine-generated code (GMAP) with error flags. Appendix B lists all possible GMAP error flag codes.

The machine-generated code is numbered in Figure 1-2 for purposes of describing the printout format and is as follows:

1. Storage location in octal.
2. Octal representation of the instruction (12 digits).
3. Octal representation of the relocation bits.
4. Label field. If there is a label, it will be the programmer's own label or a compiler generated label of the form (N)G. The term (N) is some integer value.
5. Machine operation mnemonic.
6. Operand(s).
7. Source statement line number.

```
  0002       TABLE TABA V 5 P $                          GE080220      0021
  0003         BEGIN ITEM TAA A 10 S 5 $                 GE080230      0022
  0003           BEGIN 1.0A5 2.0A5 3.00A5 4.0A5 END      GE080240      0023
  0004               ITEM TAB A 20 S 5 $                 GE080250      0024
  0004           BEGIN 1084.80A5 1084.89A5 END           GE080260      0025
  0005               ITEM TAC A 10 S 5 $                 GE080270      0026
  0005           BEGIN -8.0A5 END                        GE080280      0027
  0006       DEFINE IFEITHER ''IFEITH'' $                GE080290      0028
                                                                  000000
******ERROR..STA..0006    MISSING END
```

Figure 1-1. Syntax Example

```
0211            BEGIN WS9  = WS9 + RANGE $                          0196
0212                  WS10 = WS10 + 1 $                             0197
0213                  WS11 = (RANGE*10)/RAD-.4999 $                 0198
0214         FREA($WS11$) = FREA($WS11$)+1 $                        0199


                          Source Listing Excerpt


******ERROR....STA      0086       TOO FEW SUBSCRIPTS
******ERROR....STA      0089       TOO FEW SUBSCRIPTS
******ERROR....STA      0158       PRECLUDED BY CONTEXT
******ERROR....STA      0214       NOT TABLE ITEM
******ERROR....STA      0214       NOT TABLE ITEM
******ERROR....STA      0283       NOT TABLE ITEM


                      Semantic Errors Found in Phase Two


     (1)          (2)              (3)      (4)   (5)        (6)          (7)

    001255 | 216000   4350  03 | 000 |          | UFA |      216000,DU | 000213
    001256 | 021367   7560  00 | 010 |    8G    | STQ |      WS11       | 000213
  E 001257 | 206502   2360  00 | 000 |          | LDQ |      206502     | 000214
  E 001260 | 000010   0010  00 | 000 |          | MME |      10         | 000214
    001261 | 006400   4310  03 | 000 |          | FLD |      6400,DU    | 000215
    001262 | 021350   5650  00 | 010 |          | FDV |      RANGE      | 000215


                          Machine Code Generated
```

Figure 1-2. Usage Error Example

SECTION II

SIGNS AND CONSTANTS

SIGNS

All JOVIAL symbols are constructed by various combinations of the following signs a sign being defined as a letter, numeral, or character:

- Letters:

  A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

- Numerals:

  0,1,2,3,4,5,6,7,8,9

- Characters:

  + - * / = . , ' ( ) $ #

Arithmetic Operators

Arithmetic Operators	Definition
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

The order of operations in a given formula would be: exponentiation, multiplication and division, addition and subtraction, relational, logical.

Logical Operators

Logical Operators	Definition
AND	0 * 0 = 0
	0 * 1 = 0
	1 * 0 = 0
	1 * 1 = 1
OR	0 + 0 = 0
	0 + 1 = 1
	1 + 0 = 1
	1 + 1 = 1
NOT	/ 1 = 0
	/ 0 = 1

Relational Operators

Relational Operators	Definition
EQ	equal to
NQ	not equal to
GR	greater than
LS	less than
GQ	greater than or equal to
LQ	less than or equal to

Separators

Separators	Definition
.	decimal point in numbers
, (comma)	separate parameters
=	equate
==	exchange
$	terminate JOVIAL statement
_ (underscore)	explicit denotation of a single blank space; separating blanks may be omitted provided the omission does not join two letters, two numbers, or a letter and a number

Enclosures

Enclosures	Definition
( )	enclose parameter list
($ $)	enclose subscripts
(/ /)	enclose numeric formula producing an absolute value
(* *)	enclose numeric formula; same as **. Exponentiation.
''	enclose comments - two concatenated apostrophes
BEGIN END	bracket items

Type Descriptor

Type Descriptors	Definition
I	integer item
A	arithmetic fixed point item
F	floating point item
B	Boolean item
H	Hollerith item
T	transmission item
S	status item

Reserved JOVIAL Words

The following are reserved words and have a fixed meaning in the JOVIAL language; these words may not be used as names.

ABS	FILE	NQ
ALL	FOR	NWDSEN
AND	GOTO	ODD
ARRAY	GQ	OR
ASSIGN	GR	ORIF
BEGIN	IF	OUT
BIT	IFEITH	OVERLAY
BYTE	IN	PROC
CHAR	IO	PROGRAM
CLOSE	ITEM	RETURN
COMMON	JOVIAL	START
COMPOOL	LOC	STOP
DEFINE	LQ	STRING
DIRECT	LS	SWITCH
END	MANT	TABLE
ENT	MODE	TERM
ENTRY	NENT	TEST
EQ	NOT	TRUE
FALSE		WAIT

CONSTANTS

JOVIAL makes use of five primary types of values: numeric, boolean, literal, status, and octal.

Numeric

Numeric values may further be catagorized into integer, fixed, and floating values.

Integer values are integer numbers right-justified in one or two words.

Fixed values are decimal numbers written with or without a decimal point, followed by a decimal exponent written as the letter E, followed by a signed or unsigned integer constant. This may be followed by the letter A and a signed or unsigned integer constant to indicate the number of bits to reserve for the decimal fraction. A negative A will cause bits to the left of the decimal to be truncated. One word will give a precision of 8 digits. Two words will give a precision of 18 digits.

Floating values are decimal numbers written with or without a decimal point, followed by a decimal exponent written as the letter E, followed by a signed or unsigned integer constant. Internally, a floating value is defined as a sign bit, a 7-bit characteristic, a mantissa sign bit, and a 27-bit mantissa; i.e., a precision of 8 digits. A floating point value is always contained in one computer word.

Boolean

Boolean values are single bit, 0 or 1; 0 indicating false, 1 indicating true.

Literal

Literal values represent strings of signs: alphabetic, numeric, or special. Hollerith or transmission code may be selected. Each uses a different type of 6-bit encoding. Appendix C lists the octal representation for encoding of the signs.

The number of words necessary to contain a literal varies directly with the number of characters represented. Examples of literals are as follows:

5H(ABCDE) or 3T(ABC)

Status

Status values are members of ordered sets of unsigned integer values. A set may contain one or more members. The first member of a set is the value 0; the second, 1; the third, 2; and so forth. Values are carried as integer values and may be referenced symbolically.

Octal

Octal values represent a numeric value of 1 to 12 digits constructed by the use of the following numbers: 0,1,2,3,4,5,6,7. The numeric value may represent an unsigned octal integer, Hollerith, or transmission code.

SECTION III

VARIABLES AND FUNCTIONS

VARIABLES

Values assigned to variables may be of the following types; numeric, Boolean, literal, status.

Type variable	Assigned value
BIT	Integer
BYTE	Literal
CHAR	Integer, implicitly defined as I 8 S.
HOLLERITH	Literal
INDEX	Integer, implicitly defined as I 18 S.
LOC	Integer
MANT	Fixed
NENT	Integer
ODD	Boolean
TRANSMISSION	Literal
BASIC	Integer, floating, fixed, Boolean, Hollerith, transmission, status. The basic variable is comprised of a simple or subscripted variable.
SIMPLE	Simple item name
Subscripted	Array name ($subscript, subscript,....$) Table item name ($subscript$)
REM	Integer
REMQUO	None

Subscripts may be represented by numeric formula. If the value produced by the numeric formula is floating, it will be truncated to an integer value.

SPECIAL VARIABLES

BIT - used to manipulate bits in a defined word, usually in conjunction with numeric values.

General case

```
BIT($a$)(item name) $

BIT($a,b$)(item name($index$)) $

BIT($a,b$)(array name($c,d,e,...$)) $
```

a - initial bit

b - number of consecutive bits

c,d,e - elements of an array

Examples

```
BIT($A$)(ALPHA)=BIT($A$)(BETA) $

BIT($0,1$)(GAMMA($0$))=BIT($0$)(GAMMA($4$)) $
```

BYTE - used to manipulate 6-bit bytes from a defined word, usually a literal word.

General case

```
BYTE($a,b$)(item name) $

BYTE($a,b$)(item name($index$)) $

BYTE($a,b$)(array name($c,d,e,...$)) $
```

a - initial byte

b - number of consecutive bytes

c,d,e - elements of an array

Examples

```
BYTE($A$)(PHI)=BYTE($A$)(DELTA) $

BYTE($0,1$)(SIGMA($1$))==BYTE($1$)(SIGMA($1$)) $
```

CHAR - an integer value, implicitly declared as I 8 S, having a value between -127 and +127, inclusive. It is used as the exponent portion of a floating point value.

General case

CHAR(floating point item name ($index$)) $

CHAR(floating point item name ($a,b,c,...$)) $

a,b,c - elements of an array

Example

```
EXPONT   =   CHAR (FLOAT ($A$)) $
```

MANT - represents the significant bits of a declared floating point value. MANT has an implicit declaration of A 28 S 27 and a fixed value less than or equal to $2^{27}-1$ and greater than or equal to 2^{-27}.

General case

MANT(floating point item name ($index$)) $

MANT(floating point item name ($a,b,c,...$)) $

a,b,c - elements of an array.

Example

```
MANTIS   =   MANT (FLOAT ($1$)) $
```

ODD - represents the least significant bit of a numeric basic variable. If the basic variable is fixed or floating, its bit pattern will be treated as though it were an integer.

General case

ODD(item name($index$)) $

ODD(item name($a,b,c,...$)) $

a,b,c - elements of an array.

Example

```
IF   ODD(UNKNW($A$)) $
```

If UNKNW is odd, the statement will be true.

```
BOLEN   =   ODD(UNKNW ($A$)) $
```

If UNKNW is odd, the Boolean term BOLEN will have a value of 1.

FUNCTIONS

ABS - produces on output some unsigned numeric value representing the magnitude of a given numeric formula. The type assigned to the ABS value shall be the same as the type of the numeric formula.

General case

```
a = ABS(b) $

a = (/b/) $
```

Examples

```
ALPHA = ABS(BETA) $

ALPHA = (/BETA/) $

IF (/ALPHA*BETA/) GR 2 $
```

ALL - may be used in conjunction with a FOR statement when it is desired to process tables entry-by-entry. Tables having a rigid structure or tables having a variable structure, for which the nent words have been preset, may be processed. Processing begins with the last entry and continues until entry zero is processed (takes the form FOR I = NENT(TBL)-1, -1,0 $ for its generated code).

General case

```
FOR _ a  =  ALL(table name) $

FOR _ a  =  ALL(table item name) $
```

Examples

```
FOR A = ALL(TABNAM) $

FOR I = ALL(TIN2) $
```

ENTRY or ENT - is used to select an entire entry within a declared table and to use this specified entry in relational, assignment, or exchange statements. The structure of the selected entry is dependent on the structure and relationships of the table items making up the entry.

Caution must be used when ENTRY operations are being carried out to insure against the loss of information. The assignment of an entry value larger than the maximum defined for the value being assigned to, will cause truncation of the excess bits.

General case

```
ENTRY(table name($index$)) $

ENT(table name($index$)) $
```

Examples

```
ENTRY(TN($A$)) = 0 $

ENT(TN1($A$)) = ENT(TN2($A$)) $

ENTRY(TN1($A$)) == ENTRY(TN2($0$)) $
```

LOC or 'LOC - produces an integer number which represents the relative address (address with respect to the beginning of slave storage) of the input argument. The argument may be an item name, array name, table name, or table item name.

General case

```
LOC(name)
```

Examples

```
ALPHA = LOC(TABNAM) $

BETA = GAMMA + LOC(ALPHA) $
```

NENT - (N = number, (of), ENT = entries) identifies the NENT word of a variable length table as the NENT variable. The magnitude of the NENT variable may not exceed the declared maximum number of entries for the designated table.

General case

```
NENT(table name) $

NENT(table item name) $
```

Examples

```
NENT(TNAME) = 0 $

IF NENT(TINS) NQ 4 $

FOR A = 0, 1, NENT(TABNAM) - 1 $
```

NWDSEN - (N = number, (of), WDS = words, (per), EN = entry) an unsigned integer value representing the number of words per entry of the declared table.

General case

```
NWDSEN(table name)
```

Examples

```
ALPH1 = NWDSEN(TNAME) $

GAMMA = NWDSEN(TNAME)*NENT(TABNA) $
```

REM - generated as in-line code, yields remainder of a division of two integers. The integers may be integer constants, item names, or table names, or mode-defined items that have been defined as single-precision integer data types.

General case

```
ALPHA = REM(name,name) $
```

Example

```
ALPHA = REM(NUM,DEM) $
```

REMQUO - generated as in-line code, yields both remainder and quotient of a division of two integers. The integers may be integer constants, item names, or table names, or mode-defined items that have been defined as single-precision integers.

General case

```
REMQUO(name,name=name,name) $
```

Example

```
REMQUO(NUM,DEM=QUO,REM) $
```

SECTION IV

SIMPLE STATEMENTS

STATEMENT NAME

Names may be attached to statements that are referred to from other parts of the program. Each name is a unique, two to six alphanumeric character identifier. The first character may not be numeric and the last character must be followed by a period in the statement for which it is the name. A statement name defined within a FOR statement must not be referenced from outside that FOR statement. For example:

```
FIRST.    ALPHA = BETA * DELTA $
```

ASSIGNMENT STATEMENT

Assignment statements consist of a variable name to the left of an equal sign with an expression to the right of the equal sign. Since the expression on the right is assigned to the variable on the left and the mode of the result is that of the variable, certain rules must be set down.

1. The precision of the result will be that of the variable.

2. If by the declaration of the variable the precision is decreased, the least significant bits will be truncated.

3. If by the declaration of the variable the precision is increased, least significant bits having a zero value are added.

4. The magnitude of the expression to the right of the equal sign is not permitted to exceed the magnitude of the declared variable.

5. A negative expression must not be assigned to a variable declared to be unsigned.

6. If a declared literal variable has assigned to it a literal expression which does not agree in size, excess bytes will be truncated from the left of the expression if it is larger. For the case where the literal expression is smaller than the declared variable, blanks will be added to the left of the expression.

7. The declared variable may represent a status value, providing the expression to the right of the equal sign is an unsigned integer, compatible with the status constants of the declared variable.

8. A declared Boolean variable may have assigned to it an expression which represents an unsigned 0 or 1.

Figure 4-1 indicates valid assign statement combinations. The symbol in each block indicates the mode after the assignment has been made.

Right Side Of Equal Sign

Left Side of Equal Sign: Variable \ Right Side: Expression	Fixed Point	Floating Point	Integer	Octal	Hollerith	Trans-mission	Boolean	Status
Fixed Point	Fixed	Fixed	Fixed	Fixed				
Floating Point	Floating	Floating	Floating	Floating				
Integer	Integer	Integer	Integer	Integer				
Hollerith				Hollerith	Hollerith			
Trans-mission				Trans-mission		Trans-mission		
Boolean			Boolean				Boolean	
Status								Status

Figure 4-1. Valid Assign Statement Combination

EXCHANGE STATEMENT

The exchange statement is similar to the assignment statement. There is a left variable followed by two equal signs and a right variable. Effectively, the left term is replaced by the right term; the right term is replaced by the left term. Hence an exchange is made. Rules for the assignment statement also apply to the exchange statement.

General case

```
a == b$
```

Example

```
ALPHA($A$) == ALPHA($B$)$
```

GOTO STATEMENT

A GOTO statement allows a controlled interruption of the normal sequence in which operations are executed and causes a transfer to the statement with the name following the GOTO. The name may be a statement name, close name, or switch name.

General case

```
GOTO a $
```

Example

```
GOTO ALPHA $
```

SWITCH STATEMENT

Switch statements provide associated GOTO statements with the capability of testing various conditions and selecting specific branches to follow when a given condition exists. A switch statement defined within a FOR statement must not be referred to by a GOTO outside of the FOR statement.

The ITEM switch tests the contents of the item referred to against a number of given values and exits when the test is satisfied.

General case

```
SWITCH _ switch name (declared item name) = (first term data is tested
against = exit, ... nth term data is tested against = exit) $
```

Associated GOTO formats

```
GOTO _ switch name $

GOTO _ switch name($L$) $
```

L is a single letter variable indicating a table entry.

```
GOTO _ switch name($L,M,N$) $
```

L,M,N are components of an array.

Examples

```
SWITCH FCFIXS(CPFIXI) = (6H(OPMODE) =

FCXMOD,6H(ORBIT) = FCXRDN,6H

(GENNUM) = FCXRDN,6H(COMMON) = FCXRDN) $

GOTO FCFIXS ($FCXX$) $
```

INDEX switches use the value of the index to determine the exit to be taken. Each term in the INDEX switch exit list has an implicit value assigned. The first exit is 0, the second 1, and so forth. If no exit is provided for a given position in the list, a null term is indicated by adding the comma and continuing.

General case

```
SWITCH _ switch name   =   (exit name-1, exit name-2, ,
exit name-4,. . exit name-n) $
```

Associated GOTO formats GOTO _ switch name ($index$) $

```
GOTO _ switch name($item name ($index$) $) $

GOTO _ switch name($arithmetic expression$) $
```

Examples

```
SWITCH   BETA1   =   (B01, B02,,B04) $

GOTO   BETA1($B$) $
```

STOP STATEMENT

STOP $, or STOP_label $, is used to specify the operational end of a program or subprogram. The optional label has no effect on processing.

Execution of a STOP statement in the context of a main program, a procedure, function or close internal to a main program, or an external procedure or function will effect a return to the operating system for wrapup and termination of execution.

Execution of a STOP statement in the context of a closed subroutine, or a procedure, function or close internal to a closed subroutine will effect a return from the closed subroutine to the calling routine.

COMPOUND STATEMENT

A compound statement is identified as at least one statement included between BEGIN, END brackets. Compound statements may exist within compound statements. The set of statements making up the compound statement serve to jointly satisfy an isolated operation.

Example

```
IF   SALES   GR   QUOTA $

BEGIN ''COMPOUND STATEMENT''

   ANNUAL   =   V(STEP 1) $

   PROFIT   =   V(Percent) $

   BONUS    =   V(PLUS) $

END ''COMPOUND STATEMENT''

GOTO   REVIEW $
```

When the IF statement is true, the compound statement will be executed. A false condition would cause the compound statement to be bypassed and the next statement to be executed would be GOTO REVIEW $.

SECTION V

COMPLEX STATEMENTS

IF STATEMENT

An IF statement is a decision or conditional statement using Boolean formula, relational operators, or logical operators to provide a means of arriving at a true or false result during execution. Logical operators used in conjunction with relational operators allow a single IF statement to test several conditions. If the result is declared true, the next statement is executed. If the result is declared false, the next statement is ignored and execution continues from there.

General case

```
IF_A_EQ_B $
```

Example

```
IF   ALPHA   EQ   BETA   $
```

IFEITH STATEMENT

The IFEITH statement is similar to the IF statement with the addition of alternative statements to be executed if the initial condition is not satisfied. An END statement must follow the last ORIF clause.

General case

```
IFEITH_A_GR_B $
  A1_=_2 $
ORIF_C_LS_D $
  A2_=_1 $
END
```

Example

```
IFEITH  TEMP($0$)   GR   HOT($1$) $
T1   =   1   $
ORIF  TEMP($0$)  EQ COLD($1$) $
BEGIN
T2 = 1 $
T3 = 1   $
END
ORIF   TEMP($0$)   LS   SUBZ($1$) $
T3 = 0 $
END ''IFEITH''
```

FOR STATEMENT

In JOVIAL programming, the FOR statement provides an iteration capability when it is desirable to loop through sets of instructions, repeat execution of a single instruction, or progress through tables in either direction. There are three types of FOR statements.

1. One-factor FOR statements are used primarily as counters. The single letter variable is initialized in the FOR statement and then is added to or deleted from and tested by the programmer with each pass through the given instructions.

General case

```
FOR A = M $
```

Example

```
FOR   A = 0 $
  A = A + 1 $
IF   A   GR   10 $
```

2. Two-factor FOR statements have not only the initial value of the one letter variable but also have as a second parameter a term used as the increment or decrement of the variable. In this case, the programmer need only add a test for some maximum value.

General case

```
FOR _ A = B,C $
```

Example

```
FOR   A = 0,2 $

BEGIN ''A''
        .
        .
        .
IF    A   EQ   200 $

GOTO    EXIT $

END ''A''

EXIT. NEW = OLD + TOTAL $
```

3. Three-factor FOR statements have the same format as the one-factor and two-factor FOR statements as well as a third parameter which defines the limit or range of the FOR statement. No additional code is necessary to increment, decrement, or test for a limit.

General case

```
FOR _ A  = B,C,D $
```

Example

```
FOR   A = 0, 1, 200 $

  BEGIN ''A''
          .
          .
          .
  END ''A''
```

Each of the parameters appearing on the right of the equal sign in a FOR statement may occur as a variable, constant, or arithmetic expression.

BEGIN, END brackets must follow the declared FOR statement in each case or the simple letter alphabetic subscript will be valid only for the next statement.

Use of more than one declared FOR statement with a corresponding set of BEGIN, END brackets allows several levels of iteration to be carried on. During execution, the END bracket returns control to the statement following its matching BEGIN.

Several FOR statements may be declared in sequence preceding a single set of BEGIN, END brackets. The iteration is controlled by the first three-or two-factor FOR statement. If several three-factor FOR statements have been declared in sequence, only the first will control iteration. The others will be treated as two-factor FOR statements.

TEST STATEMENT

TEST $ or TEST _a $, where a represents a single letter subscript, provides a means of controlled interruption of operations within a two- or three-factor FOR statement.

Use of a TEST $ statement generates an implied GOTO statement directed to the operating END bracket of the innermost FOR statement (in which TEST $ is contained) that increments or decrements an index.

If TEST _a $ is used, an implied GOTO statement is directed to the operating END bracket of the FOR statement (in which TEST _a $ is contained) that increments or decrements the single letter subscript corresponding to a.

SECTION VI

DATA DECLARATIONS

ITEM DESCRIPTIONS

Item descriptions are an important part of the JOVIAL language insomuch as they give distinct features to the structure of the data being processed and manipulated by the program. The format of a declared item statement is basically:

ITEM _ item description $.

The various types of items and corresponding formats are presented in the following material.

Simple Item - is a variable that has but one occurrence and is not defined in a table. The format for a simple item is as follows:

General case

ITEM _ item name _ constant $

ITEM _ item name _ item description $

ITEM _ item name _ item description _ P _ constant $

P - indicates the item will be preset to some value

Examples

ITEM ALPHA 30.5 $

ITEM BETA B $

ITEM GAMMA A P 7 $

Fixed point item - is a variable represented by decimal digits with or without a decimal point.

General case

ITEM _ item name _ A _ size _ S/U _ precision _ R _ $

ITEM _ item name _ A _ size _ S/U _ precision _ P _
preset value $

A - indicates arithmetic

size - indicates number of consecutive bits

S/U - signed or unsigned (should be one letter)

precision - indicates number of bits after the decimal

R - rounding to the specified precision

P - preset to some value

Examples

ITEM FIX1 A 36 S 0 P 0 $

ITEM FIX2 A 19 S 9 R 17 $

Note: Unsigned fixed point item should not exceed 35 bits if only one word of storage is desired, i.e., (ITEM ALPHA I 35 U $).

Floating point item - always requires a full 36-bit word and is always signed.

General case

ITEM _ item name _ F _ R $

ITEM _ item name _ F _ P _ preset value $

F - indicates floating point

R - rounding

P - preset to some value

Examples

ITEM FLT1 F R $

ITEM FLOT F P -1.23 $

Integer item - is a signed or unsigned value without a decimal point. The decimal point is assumed to be to the right of the rightmost digit.

General case

ITEM _ item name _ I _ size _ S/U _ R _ $

ITEM _ item name _ I _ size _ S/U _ P _ preset value $

I - indicates an integer item

size - indicates number of consecutive bits

S/U - signed or unsigned (should be one letter)

R - rounding to the specified precision

P - preset to some value

Examples

```
ITEM INTE I 36 S R $

ITEM INTER I 18 U P 0 $
```

Note: Unsigned integer item should not exceed 35 bits if only one word of storage is desired, i.e. (ITEM ALPHA I 35 U $).

Literal items - describe 6-bit alphabetic, numeric, or special signs represented in Hollerith or transmission code (see Appendix C). *

General case

ITEM _ item name _ H/T _ number of 6-bit characters $

H/T - use one letter to indicate type literal

Examples

```
ITEM HOL1 H 3 $

ITEM TRAN T 18 $

ITEM HOLL H 4 P 4H(HOL1) $
```

Boolean item - is a 1-bit item used to indicate true or false conditions. If the value is 1, the item indicates a true condition. If the value is 0, the item indicates a false condition.

General case

```
ITEM _ item name _ B $
```

B - indicates a Boolean item

Examples

```
ITEM BOL B $

ITEM BOOL B P 0 $
```

Status items - provide a test panel for the program. Various conditions may be indicated by the status item. The status list contains a status quantity for each condition. Each quantity is represented by an unsigned integer value. The first is assigned 0; the second, 1; the third, 2; etc.

The size term indicates the number of bits required to represent the largest integer value assigned a status quantity. It must not exceed 71 bits. This is an optional field and the compiler will determine the size required if this field is omitted.

General case

```
ITEM _ item name _ S _ size _ V(status-1) _
    V(status-2). . . . _ V(status-n) $
```

S - indicates a status item

Examples

```
ITEM STAT S V(JAN) V(FEB) V(MAR) $

ITEM STA2 S V(JAN) V(FEB) V(MAR) $
```

MODE DECLARATION

An undeclared variable may appear in a program. If it is not preceded by a MODE description, the compiler will implicitly assign a mode of I 36 S. It is the function of the MODE description to append a definition to all undeclared variables following it. This definition will remain in effect until another MODE description is encountered or until a TERM $ is reached. A MODE description appearing in a PROCEDURE will define all undeclared variables appearing between it and the next MODE description or TERM $.

General case

```
MODE _ type of item _ size _ S/U $

MODE _ type of item _ P _ preset value $
```

type of item - A, F, B, H, T

size - indicates number of consecutive bits

S/U - signed or unsigned (should be one letter)

P - preset to some value

Examples

```
MODE A 36 S $

MODE A 36 S P 100 $
```

PRESET DATA LISTS

With the use of BEGIN, END brackets it is possible to set lists of data to given values in a format corresponding to a related declaration. If there is no related declaration, the data list values will assume the standard mode, I 36 S.

General case

```
BEGIN _ value-1 _ value-2 _ value-3 _ . . . .
      value-n _ END
```

Examples

```
BEGIN 0 1 2 3 4 5 6 7 8 9 10 END

BEGIN O(212223) O(242526) END

BEGIN V(YES) V(NO) V(YES) V(NO) END
```

DEFINE DECLARATION

The DEFINE declaration is a means of declaring an alphanumeric name which represents a given value or description. If this name is not used in a DEFINE statement again, the originally assigned description will apply throughout the program. The value or description represented may not contain a comment.

General case

```
DEFINE _ alphanumeric name _ ''value or description
       the name is to represent enclosed in
       concatenated apostrophes'' $
```

Examples

```
DEFINE LESS ''LS'' $

DEFINE STANDARD ''A 36 S'' $
```

'PROGRAM DECLARATION

The 'PROGRAM declaration serves to establish communication between the present program and another program named in the 'PROGRAM declaration and compiled independently as a closed subroutine. A 'PROGRAM declaration is a processing declaration since it names a group of statements to which control can be transferred. However, it shares with data declarations the property of not directly generating machine language coding; it can occur among the statements of a program without affecting the order of execution.

In a 'PROGRAM declaration, when transfer to a program name is specified by means of a GOTO statement, the compiler assumes that the program name is a subroutine which returns control to the statement following the invoking GOTO by means of a RETURN statement.

General case

```
'PROGRAM_Program Name $
```

Example

```
'PROGRAM SUBR $
    .
    .
    .
GOTO SUBR $
```

TABLE DECLARATION

The JOVIAL language has available two distinct table structures -- SERIAL or PARALLEL. Either structure may be selected for use with an ordinary TABLE declaration or a specified TABLE declaration. The component parts of either table structure are entries and items.

A SERIAL table is so arranged that all items which comprise an entry are located consecutively (serially). Consequently, entries as well as items are consecutively ordered.

A PARALLEL table is so arranged that all items, of each entry, having common information are located consecutively. Therefore, PARALLEL table entries are not consecutive but are located throughout the table as multiples of the number of common items.

Tables have other properties as well. They may be rigid, with a fixed number of entries specified during the table declaration. Variable tables are also allowed. The maximum number of entries the variable table may have must be indicated at the time of declaration. All tables have as the first word of the table a NENT (number of entries) word.

Items within a table may be densely packed, medium packed, or not packed at all. Dense packing may be used to conserve space. Items are packed as closely as possible in a computer word. Medium packing is used similarly to dense packing with the exception that separate items do not share the same 6-bit byte. If no packing is desired, the field is left blank when the table is declared and each item will be allocated a different computer word.

Ordinary Table Declaration

When an ordinary table declaration is made, the table items declared are arranged within an entry by the compiler. The compiler provides the word location and the bit construction of each item.

Specified Table Declaration

Table items declared in a specified table have, as parameters, terms which specifically locate the item within an entry. It is the programmer's responsibility to declare the word within the entry and the starting bit position of the item. The number of words per entry must be indicated when the table is declared.

Symbols Relating To Table Declarations

Table name - 2 to 6 alphanumeric digits

V - variable length table

R - rigid length table

Number of entries - maximum value for variable length tables; actual value for rigid length tables.

P - parallel table structure; if left blank, parallel is selected

S - serial table structure

D - dense pack items

M - treat as dense packing

N - do not pack

Number of words per entry - used in place of packing term when it is a specified table.

General case

Ordinary table declaration

```
TABLE _ table name _ V/R _ # of entries _ P/S _ D/M/N $
  BEGIN ''table''
   ITEM _ item name _ item description _ size _ S/U _ $
       BEGIN preset value/s END
  END ''table''
```

Example

```
TABLE TABEL R 7 S $
  BEGIN ''TABEL''
    ITEM TAB1 A 36 S $
      BEGIN 0 1 2 3 4 5 6 END
    ITEM TAB2 F $
      BEGIN 0 1. 2. .3E1 4.E0 19 .2E2 END
  END ''TABEL''
```

General case

Specified table declaration

```
TABLE _ table name _ V/R _ # of entries _ P/S _
      # of words per entry $
   BEGIN ''table''
    ITEM _ item name _ item description _ size _ S/U
         _ word number _ starting bit _ D/M/N $
    END ''table''
```

Example

```
TABLE ABLE R 2 S 4 $
 BEGIN ''ABLE''
   ITEM HOL H 4 0 12 $
      BEGIN 4H( JOV) 4H(IAL ) END
   ITEM TRANS T 4 1 12 $
      BEGIN 4T(LAI ) 4T(VOJ ) END
   ITEM OCT A 21 U 2 15 $
      BEGIN O(2222222) O(3333333) END
   ITEM ARITH A 19 S 9 3 17 $
      BEGIN -100.5A9 0.5A9 END
   ITEM STAT S 1 V(ON) V(OFF) 3 14 $
      BEGIN V(OFF) V(ON) END
   ITEM B00 B 3 11 $
      BEGIN 1 0 END
 END ''ABLE''
```

Tables and Subordinate Overlays

Subordinate overlays provide a means of ordering and overlaying table items within the table declaration itself. The subordinate overlay statement must appear within the BEGIN-END brackets of the table declaration.

General Case

```
OVERLAY_table item LIST $
OVERLAY_table item LIST = table item list = ...
```

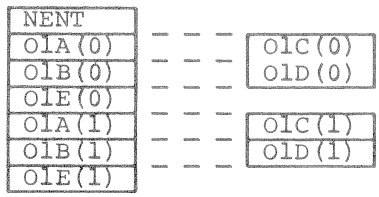
Example

```
TABLE OVLY1 R 4 S $

 BEGIN

    ITEM O1A I 36 S $
    ITEM O1B I 35 U $
    ITEM O1C H 6    $
    ITEM O1D B      $
    ITEM O1E S V(OK) V(BAD) $

OVERLAY O1A,O1B = O1C,O1D $
   END
```



STRING Table Item Declaration

The STRING table item declaration provides the means by which a programmer may describe and structure a table item. STRING permits the programmer to describe the occurrence of more than one item per table entry. Each occurrence of a STRING item is called a "bead". To refer to a particular "bead", the table item name is used, followed by a 2-component index.

General case

```
STRING_name_item_description_n3n_n4n
optional-packing-specification_n5n_n6n $
```

which can be followed by

optional-two-dimensional-constant-list

used for desired presetting of each entry of the table.

The arguments n3n and n4n indicate in which word of the entry and in which bit of the word the first "bead" of the item begins. Additionally, n5n declares the frequency of occurrence of the STRING item in the words of the entry; i.e., there are beads of the STRING in every n5nth word of the entry starting with word n3n. The last parameter of the declaration, n6n, declares the number of beads in each word of the entry.

Example

```
TABLE STR R 10 S 10 $

    BEGIN

        STRING HERA I 9 U 0 1 D 2 3 $

          BEGIN BEGIN 1 2 3 END

                BEGIN 4 5 6 END

          END
```

declares that the beads of HERA are 9-bit unsigned integers, that the first bead starts in word 0, bit 1 of the entry, that there are beads in every second word of the entry, that there are three beads in each word of the entry containing beads, and that these three beads are dense-packed (maximum compaction takes place) in the words in which they occur. The 2-dimensional constant list presets the first three beads of the first and second entries of the table.

To reference a particular bead within a table entry, a 2-component index is utilized:

```
HERA($K-3,7$) $
```

where the first component, K-3, indicates the bead within the table entry and the second component, 7, indicates in which entry of the table the bead is located.

LIKE Table Declaration

The LIKE table declaration allows, by the use of a single statement, the construction of a table similar to one which has been declared previously in the program.

If the two tables are to be identical, the LIKE table declaration is simply

```
TABLE_ original table name (X)_ L $
```

The X represents an arbitrary letter or number. This same letter or number is affixed to the end of each LIKE table item name associated with the original table item names.

It may be desirable to make a change to a given area of the LIKE table. Any change in number of entries, parallel or serial structure, or type of packing in the LIKE table declaration, will be reflected in the LIKE table composition. It is important to note that any items which are preset in the original table will not be assigned to preset values in the LIKE table.

General case

```
TABLE _ original table name (X) _ L $
TABLE _ original table name (X) _ # of entries _ L $
TABLE _ original table name (X) _ D/M/N _ L $
```

Example

```
TABLE LYKE V 4 $
   BEGIN ''LYKE''
     ITEM AA I 18 S $
     ITEM BB I 18 S $
     ITEM CC I 36 S $
     ITEM DD I 36 S $
  END ''LYKE''
```

(LIKE Table format)

```
TABLE LYKE1 L $
TABLE LYKE1 R 1 L $
TABLE LYKE1 P L $
```

The items associated with table (LYKE1) are referred to as AA1, BB1, CC1, and DD1.

ARRAY DECLARATION

ARRAY declarations provide a program with multi-dimensional arrangements of values, each having a unique position within the array. A single value occupies an element of a particular row, in a particular column of a particular plane. Those elements which do not contain some value will be said to contain a dummy value.

Elements of a column are stored in consecutive locations. The entire array is stored column by column in consecutive locations beginning with element zero, column zero, plane zero and ending with the nth element of the nth column of the nth plane.

The item description following the number of planes is a description of the values to be contained in the array. Each value must adhere to the given description.

BEGIN, END brackets must enclose the entire array, each plane and each row within each plane.

*

General case

```
ARRAY _ array name _ # of elements per column _ # of
      columns _ # of planes _ type of item _ size _
      S/U $

BEGIN ''ARRAY''

     BEGIN ''PLANE 0''

        BEGIN ''ROW 0''....preset values....END ''ROW 0''

     END ''PLANE 0''

     BEGIN ''PLANE 1''

        BEGIN ''ROW 0''....preset values....END ''ROW 0''

     END ''PLANE 1''

     BEGIN ''PLANE 2''

        BEGIN ''ROW 0''....preset values....END ''ROW 0''

     END ''PLANE 2''

END ''ARRAY''
```

Example

```
ARRAY ARAY3 3 3 3 I 36 S $
  BEGIN 'ARAY3' BEGIN ''PLANE 0''
                  BEGIN ''ROW 0'' 0 3 6 END ''ROW 0''
                  BEGIN ''ROW 1'' 1 4 7 END ''ROW 1''
                  BEGIN ''ROW 2'' 2 5 8 END ''ROW 2''
                END ''PLANE 0''
                BEGIN ''PLANE 1''
                  BEGIN ''ROW 0'' 9 12 15 END ''ROW 0''
                  BEGIN ''ROW 1'' 10 13 16 END ''ROW 1''
                  BEGIN ''ROW 2'' 11 14 17 END ''ROW 2''
                END ''PLANE 1''
                BEGIN ''PLANE 2''
                  BEGIN ''ROW 0'' 18 21 24 END ''ROW 0''
                  BEGIN ''ROW 1'' 19 22 25 END ''ROW 1''
                  BEGIN ''ROW 2'' 20 23 26 END ''ROW 2''
                END ''PLANE 2''
    END ''ARAY 3''
```

Since ARRAY data is stored column by column consecutively, the values in the example would appear in memory as 0, 1, 2, 3, 4, ..., 26.

OVERLAY DECLARATION

The OVERLAY declaration is used when it is desirable to specifically arrange declared simple items, tables, or arrays and to perform overlay operations.

It is important to consider the NENT word of a table if it is to overlay, or be overlayed, by a series of simple items or an array.

General case

```
OVERLAY _ VARIABLE LIST $

OVERLAY _ VARIABLE LIST = VARIABLE LIST = ... =
          VARIABLE LIST $

OVERLAY _ OCTAL NUMBER = VARIABLE LIST $

OVERLAY _ INTEGER NUMBER = VARIABLE LIST $
```

where a VARIABLE LIST consists of:

I/T/A name (or)
I/T/A name, I/T/A name, ..., I/T/A name

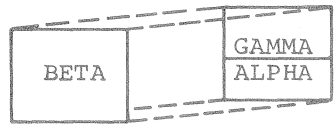
I = (simple) item name

T = table name

A = array name

Example

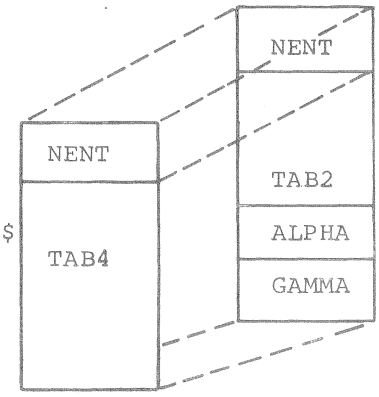
```
ITEM ALPHA I 36 S $
        .
        .
        .
ITEM BETA I 72 S $
        .
        .
        .
ITEM GAMMA I 36 S $
        .
        .
        .
OVERLAY BETA = GAMMA, ALPHA $
```



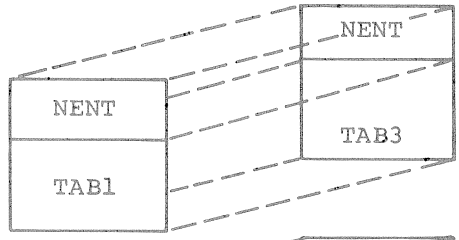

```
OVERLAY GAMMA, BETA, ALPHA $
```

```
TABLE TAB1 R 4 S $
        .
        .
        .
TABLE TAB2 R 8 S $
        .
        .
        .
TABLE TAB3 R 4 S $
        .
        .
        .
TABLE TAB4 R 10 S $
```

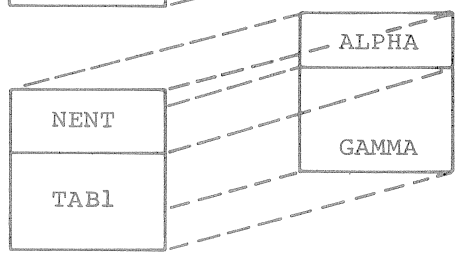
```
OVERLAY TAB4 = TAB2, ALPHA, GAMMA $
```



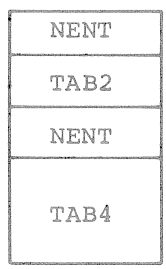
```
OVERLAY TAB1 = TAB3 $
```



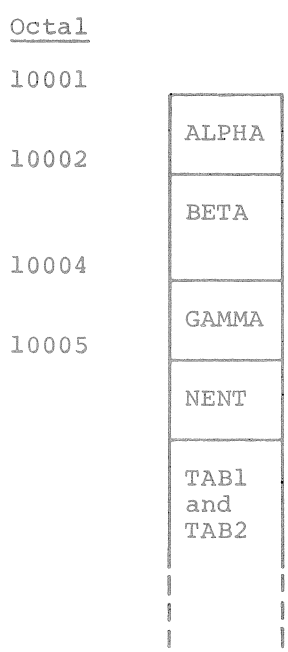

```
OVERLAY TAB1 = ALPHA, GAMMA $
```




```
OVERLAY TAB2, TAB4 $
```




```
OVERLAY O(10001) = ALPHA, BETA, GAMMA $
OVERLAY 4101 = TAB1 = TAB2 $
```



NOTES

When using an OVERLAY with an integer or octal number:

1. The integer or octal number (address) given in the OVERLAY statement must be thought of as a absolute address--relative to the base address register.

2. If the given integer or octal number (address) falls outside the limits of the program being compiled (program size), variables in the associated variable list cannot be preset (GELOAD will return an L3 abort when attempting to go beyond program length at load time). When the address falls outside the allocated core limits, the program will abort with an F0 fault if any variable in the associated variable list is referenced.

3. When specifying a double precision item as the first element of a variable list, the integer or octal number (address) must be specified as even or the item will not be positioned beginning at an even address to obtain proper accessing of data.

4. Once it appears as an element on a variable list of an OVERLAY to a fixed location, a variable cannot appear in another OVERLAY statement with proper results guaranteed.

5. Any use of this feature is potentially dangerous because the specified addresses, being relative to the base address register, may fall outside the bounds of memory allocated for the JOVIAL slave; i.e., the slave prefix and the library routine areas are accessible.

Concerning the OVERLAY statement in general, it must be remembered:

1. When double precision items are contained in a variable list, the beginning address can no longer be guaranteed even. This should be checked and changes made so that the beginning address falls in an even location for proper accessing of data.

2. All elements in variable lists of a single OVERLAY statement must be unique.

3. If an element in a variable list occurs in more than one OVERLAY statement, it must be in each occurrence after the first occurrence, the first element of the variable list in the statement.

SECTION VII

DIRECT CODE AND PROCESS DECLARATIONS

DIRECT CODE

Direct code provides the JOVIAL programmer with a means of coding in machine language while remaining in the main stream of JOVIAL code. The only restrictions while in Direct code are that no macros, no master mode instructions, and only the SYMREF, ZERO, BSS, and OCT pseudo-operations may be used. The Macro Assembler Program (GMAP) manual contains references pertaining to machine code instructions.

Direct code is considered as a complex statement and may not follow immediately after an IF statement unless bracketed by a BEGIN, END.

Index registers used while in Direct code should be saved upon entering DIRECT and should be restored prior to leaving DIRECT.

Direct code instructions may reference locations within the JOVIAL code provided the locations have user defined names of not more than six characters.

The main stream of JOVIAL code may be interrupted by a 1-word statement - - DIRECT. To re-enter the JOVIAL code after having been in DIRECT, all that is required is another 1-word statement - - JOVIAL.

DIRECT CODE RESTRICTIONS AND CONVENTIONS

All op codes must start in card column 8 and all variable fields must start in card column 16.

Alphanumeric expressions containing two or more symbols (e.g., NAME1-NAME2) are not permitted in the variable field. Expressions of the type "NAME1-5" are permitted.

Pseudo-operations:

SYMREF

The SYMREF pseudo-operation allows the user to reference symbols in Direct code which are defined outside the program environment. Thus the user may reference any of the system library routines. The routines should be accessed only with the proper argument strings. If a routine is given the same name as a system library routine, it must always be included in the load at execution time or else the loader will link to the system routine.

Only one name per SYMREF statement is allowed.

ZERO

The first field of the variable field may not be implicitly nulled.

The expression "=O" may be used to define an octal value in the variable field.

BSS

Only one numeric entry is allowed in the variable field; alphanumerics are not permitted.

Only plus (+) and minus (-) operators are permitted in numeric expressions.

OCT

Only the first entry in the variable field (delimited by the first character not an octal digit) is processed. Subsequent entries are ignored.

If the entry is less than 12 octal digits, the number is right-justified when compiled.

If the entry is greater than 12 octal digits, high-order digits are truncated.

Assign Statement

The ASSIGN statement is used only within DIRECT code. Its purpose is to assign the value of a variable to the machine accumulator (general case 1) or to assign the contents of the machine accumulator to some variable (general case 2).

If the value to be assigned to the accumulator (or from it) is Boolean, Hollerith, transmission encoding or status, the precision value should be zero.

Floating point has no precision value; however, the parentheses are used. A ( ) = V.

Fixed point or integer values have a precision value of 0-71. A zero will cause a precision of 72 bits to be assigned.

General case

```
    DIRECT
        .
        .
        .
    Machine Language
            .
            .
(1) ASSIGN _ A(P) _ = _ V($index$) $
              .
              .
    Machine Language
            .
            .
(2) ASSIGN _ V($index$) _ = _ A(P) $
              .
              .
    JOVIAL
```

A - machine accumulator

P - precision; number of bits representing the fractional portion.

V - basic variable

Example

```
Col.8   Col.16

            .
            .
            .
      ITEM NOGO B P 0 $
      IF NOGO $
      BEGIN
      DIRECT
    LDA     SIGMA
    STA     SIGMAT
    LDA     SIGMA
    ALS     6
    STA     SIGMA1
        ASSIGN A(4) = ALPHA $
        ASSIGN ALPHA = A(4) $
    LDQ     777444, DU
    ANQ     SOMVAL
    QRS     27
    QLS     1
    STQ     GONOW
             JOVIAL
         END
    IF CNT NQ O(000033) $
            .
            .
            .
            .
            .
```

CLOSE DECLARATION

It is sometimes useful to have a fixed set of statements which perform a given operation and may be accessed from several points within the main program. A CLOSE declaration is referred to by a GOTO statement directed to a particular CLOSE name. More than one CLOSE declaration may be used in a program.

A RETURN $ statement used within a CLOSE declaration will return the execution activity to the statement immediately following the invoking GOTO statement.

1. CLOSE declarations may be entered by an associated GOTO only.

2. A CLOSE declaration appearing in a PROCEDURE must be referred to only by an associated GOTO also appearing within the PROCEDURE.

3. Declared FOR statements may contain CLOSE declarations providing the associated GOTO is within the FOR structure.

4. CLOSE declarations are not recursive and may not contain any invoking statements.

5. CLOSE declarations may be nested within another CLOSE.

General case

```
GOTO _ close name $
          .
          .
GOTO _ close name $
          .
          .
CLOSE _ close name $

  BEGIN "CLOSE CODE"
          .
          .
  END "CLOSE CODE"
```

Example

```
          .
          .
          .
GOTO AAAA $
          .
          .
GOTO AAAA $
          .
          .
CLOSE AAAA $
BEGIN ''AAAA''
  IF CNT EQ 0 $
  ALPHA = ALPHA - 1 $
  RETURN $
END ''AAAA''
          .
          .
          .
```

FUNCTION DECLARATION

A FUNCTION declaration is a simple PROCEDURE declaration having only an input parameter list. During execution it generates a single output value assigned to the PROCEDURE name, which may then be used as a term in the invoking equation.

General case

```
name = formula _ function name (actual input parameters)
                          _ formula $
                          .
                          .
                          .
name = formula _ function name (actual input parameters)
                          _ formula $
                          .
                          .
                          .
PROC _ function name (formal input parameters) $
ITEM _ function name _ item description $ "DECLARATION LIST"
                          .
                          .
                          .
     BEGIN "FUNCTION BODY"
                          .
                          .
                          .
     END "FUNCTION BODY"
```

Example

```
          .
          .
          .
    IF TIME(MIN, SEC) GR 24 $

    PROC TIME(AA, BB) $

       ITEM TIME F $

       ITEM AA F $

       ITEM BB F $

         BEGIN ''TIME''

          TIME = (AA + (BB/60))/60 $

         END ''TIME''
```

1. The FUNCTION declaration PROCEDURE name must be the same as that of a declared item within the PROCEDURE declaration.

2. The value of a literal function must not exceed 12 bytes.

3. RETURN $ must not be used within a FUNCTION declaration.

PROCEDURE DECLARATION

A PROCEDURE declaration is a block of JOVIAL code which may be accessed from one or more areas within the main program. It has a specified operation to perform; input and output parameter values may be exchanged with the main program. Four major components make up a PROCEDURE declaration: procedure name, input-output parameters, declaration list, and the procedure body.

procedure name - a unique name given to a particular procedure and referred to when the procedure is to be accessed.

input-output parameters - also referred to as formal parameters. All formal parameters must correspond to a similar actual parameter. Formal simple items refer to actual simple items. Formal tables refer to actual tables, etc.

declaration list - contains declarations for all formal parameters, simple items, table names and array names.

procedure body - is made up of the code necessary to utilize the formal parameters and provide the PROCEDURE with a capability of performing a given operation.

A RETURN $, if used within a PROCEDURE, has the effect of restoring the registers, setting the output parameters, and causing an exit to the statement.

General case

```
                              .
                              .
                              .
    procedure name(actual input parameters = actual
                   output parameters) $
                              .
                              .
    procedure name(actual input parameters = actual
                   output parameters) $
                              .
                              .
    PROC _ procedure name(formal input parameters =
                          formal output parameters) $

    ITEM _ item name _ item description $ ''DECLARATION LIST''
                              .
                              .
                              .
      BEGIN ''PROCEDURE BODY''

                  .
                  .
                  .
      END ''PROCEDURE BODY''
```

Example

```
    ALPHA(A1 = A2) $
        .
        .
    ALPHA(A1 = A2) $
        .
        .
  PROC ALPHA(MPENT = PREXIT) $
    ITEM MPENT H 12 $
    ITEM PREXIT I 36 S $
    ITEM AAA I 6 U $
    ITEM BBB B $
      BEGIN ''ALPHA''
        PREXIT = 0 $ AAA = 0 $
        FOR A = 66, -6, 6 $
        FOR B = 0,1 $
          BEGIN ''FOR A, B''
            BIT($0,6$)(AAA) = BIT($ Y, 6$)(MPENT) $
            IF AAA EQ 0 OR AAA EQ O(20) OR AAA EQ O(60) $
            TEST $
            IF AAA EQ O(52) $
              BEGIN ''IF TRUE''
               BBB = 1 $ TEST $
               END ''IF TRUE''
      PREXIT = PREXIT + AAA * 10 ** B $
      END ''FOR A,B''
    IF BYTE($0$)(MPENT) EQ 1H(-) $ BBB = 1 $
    IF BBB $
    PREXIT = - PREXIT $
  END ''ALPHA''
```

1. The declaration list and procedure body must not contain another procedure declaration.

2. The procedure body must not contain a statement which directly or indirectly invokes this procedure declaration (no recursion).

3. The procedure name must not appear as a simple item in the declaration list.

Interprocedure Communication

When the attributes of a procedure or function are available, argument validation and conversion, and function result conversion will be automatically performed for each invocation. These attributes will be known when the procedure or function is defined within the same compile unit as the calling routine, or when there exists a COMPOOL declaration for the procedure or function.

A correspondence is made between actual and formal arguments. If the number of input and/or output arguments does not agree, the call is rejected with a diagnostic. Where actual and formal parameters are correspondingly table names, file names, close names, or statement names, the actual arguments are passed without modification. Where formal input parameters corresponds to actual formulae, variables, array names, or simple item names, the matching parameters are processed according to the rules for assignment statements. (Valid combinations are indicated in Figure 4-1; substitute the words 'formal parameter' for 'variable' and 'actual parameter' for 'expression'.) If any correspondence is invalid, the call is rejected with a diagnostic. If all are valid, then conversion and/or scaling will be performed as required, and the resultant actual arguments passed to the procedure or function. Where actual and formal output parameters are correspondingly variables, array names, or simple item names, the actual arguments are passed without modification. If they do not correspond the call is rejected and a diagnostic issued.

Formal input parameters which are declared as simple items follow conventions for pass by value; all other parameters follow pass by name conventions. Thus, redefinition of a simple formal input item will have no affect on the corresponding actual input parameter.

Function results are validated with respect to their use in a formula or as a right of equals operand. If the result is valid in context, then conversion and/or scaling will be performed as required.

It is permissible to access external subroutines from a routine coded in JOVIAL. The external subroutine may be coded in JOVIAL, FORTRAN, or GMAP.

System Supplied Subprograms

It is also possible to access system supplied subprograms from a JOVIAL routine. A complete list of subprograms available to FORTRAN programs is given in manual Series 6000 FORTRAN. Many of these subprograms can be readily used by JOVIAL programs. Procedure type subprograms which are particularly useful in JOVIAL include DUMP, PDUMP, LINK, LLINK, JEXIT, and CNSLIO. Function type subprograms include EXP, ALOG, ALOG10, SIN, COS, TANH, SQRT, ATAN, and ATAN2.

The following illustrates the use of DUMP and PDUMP:

System subroutines DUMP and PDUMP may be called directly from a JOVIAL coded program.

DUMP - causes the indicated limits of core storage to be dumped and execution to be terminated.

PDUMP - causes the indicated limits of core storage to be dumped and execution to be continued.

General case

```
DUMP(start location, stop location, dump format) $

PDUMP(start location, stop location, dump format) $
```

dump formats

0, dump in octal

1, dump as integer

2, dump as real

3, dump as double precision (not valid in JOVIAL).

4, dump as complex (not valid in JOVIAL).

5, dump as logical

Example

```
DUMP(ALPHA,GAMMA,2) $

PDUMP(ALPHA($0$),ALPHA($20$),0) $
```

1. If no arguments are given, all of core storage is dumped in octal.
2. If no dump format is given, it is assumed zero and the dump will be octal.

Figure 7-1 illustrates a JOVIAL main program which calls subprograms written in JOVIAL, FORTRAN, and GMAP. Control cards which might be required to compile and/or execute such a combination are not shown. Refer to the manual, Control Cards, for descriptions of control cards.

Closed Subroutines

A closed subroutine is a parameterless procedure that is callable, and which returns to its caller via STOP statements in the procedure body or in the body of a nested procedure, or via the TERM statement for the closed subprogram body. In terms of Series 600/6000 programming concepts, it is a named main program which can return to an invoking (sub) program. When invoked by the system, return is synonymous with termination.

The PROGRM directive may be used to name the closed subroutine. An unnamed closed subroutine is given the name of 'NONAME' by default. A closed subroutine named name may be invoked (called) in one of the following two ways:

GOTO name $	Where name is declared as a prime program.
LINK(6H(lkname))$	Where lkname is the name of the link overlay containing name as its entry point. lkname and name may be identical symbols.

```
PROGRM JOV
    START
          .
          .
        JOVPRO(R=S) $
          .
          .
        PDUMP(L, P, 1) $
          .
          .
        FORPRO(A, B, C = D,E,F) $
          .
          .
        GMAPRO(=X, Y, Z) $
          .
          .
        TERM $
```

```
PROCED JOVPRO

            START

            PROC  JOVPRO(U=V) $
          .
          .
    RETURN $

    END

    TERM $
```

```
          SUBROUTINE FORPRO(X, Y, Z, XX, YY, ZZ)
                        .
                        .
                        .
                RETURN

                END
```

```
             SYMDEF GMAPRO
    GMAPRO   SAVE 1,5,6,7
                .
                .
                .
             RETURN GMAPRO
             END
```

Figure 7-1. Sample JOVIAL Program With Mixed Subroutines

SECTION VIII

COMPOOL

COMMUNICATIONS POOL

One of the requisites of a programming language intended for large scale data processing systems is that it include the capability of designating and manipulating system data. The Communications Pool (COMPOOL) facility of the Series 600/6000 JOVIAL meets this requirement.

A COMPOOL serves as a central source of data and procedure descriptions, facilitating the communication of changes in design by supplying the compiler with the current descriptive parameters, thus allowing automatic modification of references through recompilation.

The COMPOOL facility is accomplished through use of a COMPOOL file. The file may reside on tape or mass storage and contains a table or dictionary of names and associated definitions for use by a system of related programs. If a program is to be entered into the system, the descriptions and locations of common data, procedures, functions, and programs are found on the COMPOOL file.

If a program written in JOVIAL makes reference to a name defined in the allocated COMPOOL file, and if this reference is compatible with the COMPOOL definition, then the reference is taken to be a reference to the COMPOOL defined name. If, however, the program explicitly and properly defines such a name (one which is defined in the COMPOOL) this definition takes precedence and the COMPOOL definition is disregarded. "Proper" definition has reference to the necessity of placing declarations ahead of references.

COMPOOL STRUCTURE

The COMPOOL is a file containing binary recorded computer representations of descriptions of simple items, table items and/or strings, arrays, tables, files, external programs, procedures and functions. These descriptions will include the following information:

1. item - name, location, item declaration information.
2. table - name, location, table declaration information including name, location and declaration information for subordinate items and strings.

3. array - name, location, array declaration information.

4. file - name, file declaration information.

5. external program - name, location

6. procedure - name, location, formal parameter list with a description of each parameter.

7. function - name, location, formal parameter list with a description of each parameter, function type.

Each name in the dictionary must be unique. Location information on the COMPOOL is a function of whether the name is of data or subprogram. When the name describes a program, procedure or function, the location information will consist of linkage information for the automatic inclusion and communication with the appropriate subprogram. When the name describes a datum, the location information will be sufficient to effect the assignment of actual core storage in one of two ways. If the name is defined relative to:

1. An unnamed COMPOOL region - storage will be allocated internal to the program being compiled, as required.

2. A named COMPOOL region - storage will be allocated after compilation. The datum has an associated relative location within the named region such that references from different program units address the same actual core storage location.

Unnamed and named COMPOOL regions are discussed in the paragraph "COMPOOL ASSEMBLY" below, along with a thorough discussion of the storage allocation algorithm of the Series 600/6000 General Loader.

ACCESSING OF COMPOOL DEFINITIONS

Use of the COMPOOL facility requires the allocation of a specific COMPOOL file at compile time. Following the $ JOVIAL control card, and prior to any ensuing control cards which delimit activities (e.g., $ JOVIAL, $ FORTRAN, $ EXECUTE) there must be a file allocation card, attaching the COMPOOL file with a file code of '.L'. Following are two examples of such allocation cards:

```
$ TAPE .L,A1D,,6341,,MY-COMPOOL
$ PRMFL .L,R,,USERID/COMPOOL4
```

Inclusion of a COMPOOL definition may be explicit or implicit. Explicit inclusion is accomplished through use of the COMPOOL statement. This statement may appear in the definitions section of the source program; it is followed by a list of names bracketted by a BEGIN/END. Upon encountering this statement, the compiler opens the .L file and institutes a search for all names. Associated definitions are extracted from the file and incorporated into the current program.

Implicit inclusion is accomplished, through reference to COMPOOL names, where no prior declaration has occurred. The COMPOOL file, if present, is searched each time a new name is encountered in a usage context (i.e., nondeclarative). If a definition exists on the COMPOOL for that name, it is accepted. If no definition exists on the COMPOOL, or if there is no COMPOOL file allocated, MODE definitions will be applied to the undefined names.

Explicit inclusion offers a compile time efficiency advantage. The COMPOOL file need only be searched once if all required names are listed. Implicit inclusion offers the advantage of user convenience, at the expense of less efficient compilation.

In any given program unit (compilation), it is unlikely that all COMPOOL names will be referenced. Considering that the names of a COMPOOL may be divided into named regions, it is also possible that not all named regions will be referenced. The program unit is compiled such that only those named regions required to support the referenced names will be defined. In a subsequent loading process then, the amount of core allocated for Communications Pool storage will be kept to a minimum.

COMPOOL ASSEMBLY

The COMPOOL assembly process involves the conversion of symbolic descriptions of simple items, tables, table items and/or strings, arrays, files, external programs, procedures and functions, the ordering of all names, and the generation of a COMPOOL file. The symbolic format used for such descriptions is natural JOVIAL. The assembly process is in fact a special kind of JOVIAL compilation.

The first card of a COMPOOL aseembly activity must contain the word GENCOM in columns 1 through 6. This serves as a signal to the JOVIAL compiler indicating COMPOOL assembly. For this activity, allocation of a COMPOOL file is mandatory; the file code '.L' is used. Following are two examples of allocation control cards:

```
$ FILE .L,X3S,25L
$ PRMFL .L,W,,USERID/COMPOOL4
```

During a GENCOM activity, first phase error checking is performed. If errors exist, an output listing is generated showing where the errors occurred and what kind of errors were made, and the compilation is terminated; the COMPOOL file is not written. When the first phase runs error free, the COMPOOL file (.L) will be written.

A limitation exists on COMPOOL names. Names which identify data and which are 11 characters or less will be fully unique. Data names which are 12 or more characters in length and which are not unique within the first 11 must have a unique length to distinguish them from other data names beginning with the same 11 characters. Thus, NAME1 and NAME2 are unique, VERYLONGNAME1 and VERYLONGNAME2 are not, but VERYLONGNAME3 and VERYLONGNAME30 are.

External program, procedure, and function names must be unique within the first six characters.

Another, more obvious, restriction applies to GENCOM activities: only data and process declarations are permitted. This includes such language features as DEFINE, OVERLAY, PROC, and 'PROGRAM. Procedures and functions declared on the COMPOOL are coded with a null process body.

Data declarations for a given COMPOOL file may be divided and grouped into named COMPOOL regions. Regions are named in one of two ways: on the GENCOM directive card or via a special naming statement, applicable only in GENCOM activities, the COMMON statement. The region name on the GENCOM card is optional. If COMMON statements are present, each must have a region name. All region names must be unique with respect to each other and with respect to other data and process names, and are restricted to six characters.

Allocation of storage for data declared under an unnamed region will be done at compile time and within internal process storage for any and all program units referencing such names. Storage will only be allocated for referenced names.

Determination of relative locations for data declared under a named region is done at COMPOOL assembly time. Allocation of storage for referenced regions (regions referenced by virtue of references to associated data names) is done at load time, prior to execution.

Named regions are compiled as Labeled Common storage. As such, the output of a GENCOM activity may include a BLOCK DATA object module. One such module will be generated for each named region, hard copy of which will be punched if the DECK option is specified on the $ JOVIAL control card.

Labeled Common is a common data area allocated by the General Loader and given a specific name. The name, which the user supplies during his COMPOOL generate activity, and the size of the data region needed for COMPOOL data, are placed on the object file of any program referencing data declared for that Common. When programs referencing a Labeled Common are loaded for execution, the BLOCK DATA subprogram object file generated for that Common must also be present if the COMPOOL contains preset information or rigid tables. Where this is not the case, the BLOCK DATA object module need not be present.

When the Loader encounters the Labeled Common name on either the BLOCK DATA subprogram object file or the object files of any programs referencing the Common, it checks the load table for its definition. If this Labeled Common has not been previously defined, it is assigned an amount of storage equivalent to the size specified (on the object file where encountered), beginning with the next available even memory location. An entry is then put in the load table for this Labeled Common name. If the Labeled Common name is found in the load table, the existing allocated area will be used (shared). This feature allows a BLOCK DATA subprogram and many COMPOOL utilizing programs to define the same Labeled Common area and all will reference this one area at run-time.

When the BLOCK DATA suprogram is loaded, preset information is stored into the specified Labeled Common region at the relative locations specified with the presets. If no preset information exists in a Common or if, for a given load, the user does not want the information preset, the BLOCK DATA subprogram object file need not be present at load time. Information contained on the object files of programs referencing a COMPOOL is sufficient to allocate storage for any required Labeled Common region but not to preset it. References to Common data are compiled as offsets relative to the Labeled Common region; this offset being carried on the COMPOOL file as the location of data.

The BLOCK DATA subprogram, required for COMPOOL's utilizing the preset feature, may be introduced into the load stream from these sources:

1. It may be the product of a COMPOOL assembly activity preceding the load activity, in which case the compiler will forward the object module on B*.

2. It may be on an object (subroutine) library. The General Loader has facilities for user libraries via the $ LIBRARY control card; the standard system library may be supplemented by a *L file; or it may be on the standard library (L*) itself.

3. The object deck for the BLOCK DATA subprogram may be included in the job stream. This inclusion may be physical, or the $ SELECT capability may be used to extract the deck from a PERM FILE, inserting it into the stream (R*).

Depending upon applications, the user may desire to utilize a given COMPOOL with different preset information in different runs. To obtain this facility, the user need only assemble the COMPOOL as many times as there are different sets of presets, and save the BLOCK DATA subprogram object files from each generation. Then, for a given run, he uses one of the options described above to include the BLOCK DATA subprogram object files which contain the desired presets.

The COMMON statement enables the specification of multiple Labeled Common regions for a single COMPOOL. Each Common statement is followed by a series of declarations enclosed within BEGIN/END brackets. Allocation of storage for these declarations is relative to the named common data space. Declarations appearing outside the brackets for any COMMON statements are enclosed within the START/TERM brackets of the GENCOM directive. Hence, allocation for these declarations is either relative to the Labeled Common named in the GENCOM card or relative to local storage if no name is supplied.

The following illustrates the organization of COMPOOL source for a single common data area:

```
GENCOM  POOL1
START
   .
   .  } declarations
   .
TERM $
```

The following illustrates the organization of COMPOOL source for several common data areas, and local storage:

```
GENCOM
START
   .
   .  } local declarations
   .
COMMON POOL2 $
BEGIN
   .
   .  } declarations for Labeled Common POOL2
   .
END
   .
   .  } more local declarations
   .
COMMON POOL3 $
BEGIN
   .
   .  } declarations for Labeled Common POOL3
   .
END
   .
   .  } more local declarations
   .
TERM $
```

Usage of Labeled Common for COMPOOL space introduces a broad functional capability to Series 600/6000 JOVIAL. Control cards may be used to override the default load/allocation algorithms. COMPOOL space can be shared with FORTRAN and COBOL modules. In short, this choice insures that the implementation is well integrated into the general software complex.

COMPOOL DISASSEMBLY

The COMPOOL Disassembly program will process a COMPOOL file, convert the encoded information, and print the contents in a variety of reports. The program, named DISCOM, is a JOVIAL utility invoked by a control card of the form:

```
$  PROGRAM DISCOM
```

Upon input of a COMPOOL file, DISCOM will generate sorted listings describing all COMPOOL names, their descriptions and the particular Labeled Common to which they belong. Listings included are: All identifiers regardless of type, Simple Items, Files, Arrays, Procedures and Functions, Subscripted Items, Tables with their Subscripted Items.

The COMPOOL file, which must be allocated for proper execution, is assigned to file code '01'. The following illustrates back-to-back assembly and disassembly of a COMPOOL file:

```
$  SNUMB ...
$  IDENT ...
$  JOVIAL
$  FILE    .L,A1S,20L
GENCOM
     .
     .    } COMPOOL Declarations
     .
   TERM $
$  PROGRAM DISCOM
$  FILE    01,A1D
$  ENDJOB
```

SAMPLE CODING

Figure 8-1 illustrates coding for the assembly of a simple COMPOOL file on tape. Only one named region is defined, POOL1. Figure 8-2 illustrates coding for use of the tape.

```
$  SNUMB . . . .
$  IDENT . . . .
$  JOVIAL
$  TAPE .L,A1D,,,,COMPOOL-TAPE
$  INCODE  IBMF
GENCOM POOL1
START
    TABLE CDN1 V 50 S 1 $
       BEGIN ''CDN1''
          ITEM AA F 0 0 $
          BEGIN 0.0 END
       END ''CDN1''
    TABLE CDN2 R 24 S 3 $
       BEGIN ''CDN2''
          ITEM BB I 6 U 0 0 $
          BEGIN 0 END
          ITEM CC A 4 S 1 1 0 $
          BEGIN 0.0 A1 END
          ITEM DD H 6 2 0 $
       END ''CDN2''
   ITEM CDN3 I 18 S P 0 $
   ARRAY CDN4 3 3 3 I 18 S $
   ITEM CDN5 B $
TERM $
$  ENDJOB
***EOF
```

Figure 8-1. Generating A COMPOOL Tape

```
$  SNUMB . . . .
$  IDENT . . . .
$  JOVIAL
$  TAPE  .L,A1D,,1234,,COMPOOL-TAPE
$  INCODE IBMF  (Not required if source is in
                Honeywell format)
PROGRM PROGA
START ''PROGA''
   ITEM NAME H 6 $
   COMPOOL
      BEGIN
        CDN1 CDN2 CDN3
        CDN4 CDN5
      END
   TABLE CDN1A L $
      IF CDN5 $
          BEGIN ''BOOL TEST''
             TEST1 $ ''CALL SUBROUTINE''
             TEST2 $ ''CALL SUBROUTINE''
          END ''BOOL TEST''
TERM $
$ ENDJOB
***EOF
```

Figure 8-2. Use Of The COMPOOL Tape

SECTION IX

INPUT AND OUTPUT OPERATIONS

FILE OPERATIONS

The following definitions define symbols used to explain the FILE, IO, IN, OUT and WAIT statements below.

fn - a declared file name of this file.

fsc - file status constants, each representing a fixed condition.

fc - file code; is always preceded by an R and numeric code between 01 and 77 or an integer variable representing a value between 01 and 77. Files 05 (I*) and 06 (P*) are reserved for system files. (Files 41, 42, and 43 are system reserved files.)

IOn - n represents any 1- to 4-digit integer value which gives each IOn statement a unique identification.

ac - action code, numeric value representing the action to be taken pertaining to this file.

ta - terminal action; either a procedure name or zero. If a procedure name is used, it must not have parameters. The procedure will be executed upon completion of the operation initiated by the IOn statement. A zero indicates no associated procedure will follow the completion of the IOn. In the current implementation, ta must equal zero.

dn - declared name of an item, table, or array where data is stored. Table name ($index$) and NENT (table name) are also valid parameters.

np - number parameter; a numeric value representing the number of words or entries in the declared data storage area.

FILE STATEMENT

All files must be declared in a FILE statement prior to any reference to the file. The FILE statement serves to define a file by name and file code. It also allows a list of status values to be appended to the file for testing. The initial status value represents an unsigned integer value of 0; the second a value of 1; the third a value of 2; etc.

The numeric value and its associated meaning for each of the file status constants (fsc) are as follows:

Numeric value	Meaning
0	Normal termination after I/O.
1	Null
2	Segment mark. Read only-- a 1-character record other than octal 17 or 23 was encountered.
3	End-of-file. A 1-character record (octal 17 or 23) was encountered, or the file has been closed by an OUT statement.
4	Buffer length error. Input area smaller than logical record.
5	Parity error.

General case

```
FILE _ fn _ V(fsc-1) _ V(fsc-2) _ V(fsc-3) _
  . . . . .V(fsc-n) _ Rfc $
```

Examples

```
FILE REDER V(NORM) V(NULL) V(SEG) V(EOF) V(BUF)
  V(PAR) R05 $ ''05-SYSIN''

FILE PRNT V(NORM) R06 $ ''06-SYSOUT''
```

IOn STATEMENT

IOn statements permit the programmer to manipulate external storage devices or transmit data to or from declared files. IOn statement action codes are placed into two categories - - transmission and device manipulation.

Transmission:

Action Code	Meaning
0	Read logical record.
1	Read segment up to the next one character record, not octal 17 or 23. Transmit data to storage.
2	Null.
3	Transmit print image.
4	Write logical record. (File and Record Control automatically partitions logical records which are longer than the size of the output buffer.)
5	Punch binary record.
6	Punch BCD record.
7	Write segment mark. A 1- character record, not octal 17 or 23, 75, 76, or 77.

General case

```
IOn(ac,fn,ta,dn,np) $
```

Examples

```
IO10(0, REDER,0,CARD,14) $ ''READ LOGICAL RECORD OF 14 WDS''

IO20(3, PRNT,0,CARD,14) $ ''PRINT LOGICAL RECORD OF 14 WDS''
```

Device Manipulation:

Action Code	Meaning
8	Move forward one logical record.
9	Move forward past segment mark or end-of-file mark.
10	Move backward one logical record.
11	Move backward over segment mark.
12	Null.
13	Rewind.

General case

```
IOn(ac,fn,ta) $
```

Example

```
IO100(13,TAPE,0) $ ''REWIND TAPE''
```

Defined combinations of a declared name and a numeric value representing the number of words or entries are listed in the following table, "IOn Statement Defined Combinations".

Declared Name	Number of words	Words Trans-mitted	Entries Trans-mitted	NENT Word Trans-mitted	Used with Serial Structure	Used with Parallel Structure
Simple Item Name	n > 0	n				
Array Name	n > 0	n				
Table Name	Absent		Declared Number	No	Yes	Yes
Table Name	n > 0		n	No	Yes	No
Table Name ($subscript$)	n > 0		n	No	Yes	No
Table Item Name ($subscript$)	n > 0	n		No	Yes	Yes
NENT (table name)	Absent or 0			Yes	Yes	Yes
NENT (table name)	n > 0		n	Yes	Yes	No

IOn Statement Defined Combinations

IN STATEMENT

The IN statement identifies a declared file which shall act as an input file. The action code portion of the IN statement explicitly defines the action to be performed on the file.

Action Code	Meaning
1	Open input file.
2	Close input file.
3	Null.
4	Close input file with rewind.
5	Release input reel.
6	Close input file and open output file.

General case

```
IN(ac, fn) $
```

Example

```
IN(1, REDER) $ ''OPEN READER FILE''
```

OUT STATEMENT

The OUT statement identifies a declared file which shall act as an output file. The action code portion of the OUT statement explicitly defines the action to be performed on the file.

Action Code	Meaning
1	Open output file.
2	Close output file.
3	Close output file with rewind.
4	Force output end-of-reel.

General case

```
OUT(ac, fn) $
```

Example

```
OUT(1, PRNT) $ ''OPEN PRINTER OUTPUT FILE''
```

Figure 9-1 illustrates a portion of a program containing input/output coding.

WAIT STATEMENT

For compatibility with the parent I/O set (NAVCOSSACT J3X), the WAIT statement will compile; however, there is no code generated and, in fact, none is needed due to the technique of interfacing with the standard I/O routines.

General case

```
WAIT (IOn) $
```

Example

```
WAIT (IO9) $
```

```
        .
        .
        .
TABLE ALPHA R 100 S $
   BEGIN ''ALPHA''
       ITEM DATA H 6 $
   END ''ALPHA''
         .
         .
         .
   FILE PRT V(YES) R 06 $
        OUT(1, PRT) $
           FOR A = 1,2,100 $
            BEGIN ''FOR A''
        IO1(3,PRT,0,DATA($A$),14) $
               END ''FOR A''
                     .
                     .
                     .
                     .
        OUT(2, PRT) $
ENDALL. STOP $
        TERM $
```

Figure 9-1. Example Of I/O Print Information

SECTION X

OBJECT CODE DEBUG

DEBUG OPTION

To initiate object code debug, the DEBUG option must be selected on the $ JOVIAL control card. From that point on, the debug procedure is automatic. The JOVIAL compiler analyzes the source program as it is being compiled to determine the importance of variables, to recognize loops, and to understand the general flow of the program. Using this information, it inserts code to provide for:

1. Dumps of key variables, such as FOR, assignment and I/O variables, as they change.
2. A program trace, which is a list of program transfers.
3. A listing of procedure entrances and exits.

All debug information starts with the name of the program. The main program is signified by an asterisk. Each statement trace starts with the compiler-assigned statement number followed by a slash mark. In an assignment statement, this is followed by the variable name (subscripted if necessary) and the value assigned to it. For the FOR statements, the value of the induction variable is given for each iteration.

Tracing automatically terminates after each statement has been executed n times, where n is a system parameter. The format is (1) line number, (2) OFF, (3) the number of statements executed prior to termination, and (4) the number of statements executed during the trace-off mode. Automatic tracing resumes when statements which have not been previously executed n times are encountered.

IF, IFEITH and SWITCH statements are followed by the values of their expressions. The final debug information supplied under post-mortem consists of the results of the last three executions of each statement. The format is (1) line number, (2) number of times the statement was executed, and (3) the last three values. Statements not executed three times show values in accordance with the number of executions.

The DEBUG feature is described in detail in the General Loader manual.

SECTION XI

CONTROL CARDS

All control cards that may be required for processing a JOVIAL program are described in detail in the Control Cards Reference Manual. Those control cards which are fundamental to the program are described briefly below.

$ SNUMB

1	8	16
$	SNUMB	Job identifier, urgency

The $ SNUMB control card is used by System Input to identify the job internally and to assign an urgency to the job for use by System Input in allocating the job. The job identifier is from one to five characters in length and must be present on the card. The urgency level is a number from 1 to 63 (at present 40 is ''threshold'' level) and represents the relative importance of the job. If the urgency is omitted, a value of 5 is assumed.

The $ SNUMB card must be the first card of every job. The job identifier can be alphabetic, numeric, or alphanumeric. The job identifier must not contain all zeros. SYSOUT interprets an all-zero SNUMB as error and the job will be ignored.

$ IDENT

1	8	16
$	IDENT	Account No., Identification

The $ IDENT control card is used to identify the user of a job or activity and to supply accounting information. Each activity may be preceded by a $ IDENT card or a single $ IDENT may be used for a series of activities.

The format of the operand is an option of the individual installation. However, it is recommended that the first field not contain any alphanumeric account number exceeding 12 characters. The operating system scans the field looking for a comma and then takes the next nine columns to create a banner for the execution report. If no comma is found before column 52, columns 52 through 60 are used for the banner. At least one $ IDENT must immediately follow the $ SNUMB card.

$ OPTION

1	8	16
$	OPTION	Options

The $ OPTION control card is used to alter the standard loader options during loading. It is placed after the $ IDENT card.

The operand field may contain one or more of the following options. Standard options (those underlined) will be used when options are not specified.

MAP - A memory map will be produced.

NOMAP - No memory map will be produced.

CONGO - The job is executed regardless of any nonfatal errors detected during loading.

GO - The job is executed only if no errors, fatal and nonfatal, are detected during loading.

NOGO - The job is not executed after loading. When loading is completed, a slave memory dump is taken.

SET/N/ - Allocated memory is set to the octal pattern specified in (N) (memory is normally set to zero). The number (N) may be any octal pattern up to 12 octal characters ((N) will be right-justified with leading zeros).

ERCNT/N/ - The number of fatal and nonfatal error messages, which are printed, is limited to a total number (n) before loading is aborted. The limit is normally set to 150.

SYMREF - SYMREFs used in each subprogram loaded are listed in the memory map. This option may be set or reset at any time during loading.

NOSREF - No SYMREFs are printed.

JOVIAL - This option sets all options required for loading programs generated by the JOVIAL compiler.

1. The options must be separated with commas.

2. The JOVIAL option must be specified for loading and executing JOVIAL object decks.

3. Options may appear on the $ OPTION control card in any sequence.

4. The $ OPTION control card must precede the object deck for which the option is to take place.

5. For programs which include a mixture of JOVIAL with FORTRAN and/or COBOL modules, the elected option should correspond to the language used for the main program. For example, if a FORTRAN main program calls upon JOVIAL subprograms, $ OPTION FORTRAN is appropriate; if a JOVIAL main program calls upon a FORTRAN subprogram, $ OPTION JOVIAL is required.

$ JOVIAL

```
1    8          16
$    JOVIAL     Options
```

The $ JOVIAL control card is used to call in the JOVIAL compiler. The operand field may contain one or more of the following options. Standard options (those underlined) will be used unless non-standard options are specified.

DECK - Produce a binary object deck for later execution.

NDECK - Do not produce a binary object deck.

LSTIN - Produce a listing of the source program.

NLSTIN - Do not produce a listing of the source program.

LSTOU - Produce a machine language (GMAP) listout of the compiled program.

NLSTOU - Do not produce a machine language (GMAP) listout of the compiled program.

COMDK - Produce a compressed source deck of the input program.

NCOMDK - Do not produce a compressed source deck of the input program.

DEBUG - Produce a Debug Symbol Table for optional use by the system loader at execution time.

NDEBUG - Do not produce a Debug Symbol Table.

SYMTAB - Produce a Set-Used Listing at compile time that is constructed to indicate the occurrence of all program names, their statement numbers, and whether the names are declared or used in particular statements.

NSYMTAB - No Set-Used Listing is produced.

DUMP - Dump slave core if activity terminates abnormally.

NDUMP - Dump program registers, upper SSA, and slave program prefix if activity terminates abnormally.

1. The options must be separated with commas.

2. Options may appear on the $ JOVIAL control card in any sequence.

3. The $ JOVIAL control card must immediately precede the JOVIAL source program deck. If several JOVIAL source programs make up the job, each must be preceded by a $ JOVIAL with the desired options for each program.

4. If the source deck is punched using the IBM character set, a control card, $ INCODE, must be placed between the $ JOVIAL control card and the first card of the source deck. For FORTRAN, the operand field is IBMF; for COBOL, the operand field is IBMC.

$ EXECUTE

1	8	16
$	EXECUTE	Option

The $ EXECUTE is used to request the loading of an object program.

The operand field may contain the word DUMP, which will force a memory dump of the program if the execution does not terminate normally. The $ EXECUTE control card must appear after the main program and all subprograms to be executed but prior to their data.

$ LIMITS

1	8	16
$	LIMITS	Time, Storage 1, Storage 2, Print Lines

The $ LIMITS control card is used to extend standard activity limits.

Time - Specifies the maximum run time for the execution activity expressed in hundredths of an hour. If the specified time is exceeded, the job is aborted. The maximum value that may be used in this field is 999.

Storage 1 - Denotes the maximum core storage requested for executing this job. The units of storage are decimal digits representing the number of words desired. (Actual memory allocation is made in multiples of 1024 words.) The smallest request that will be allocated is 1024 words.

Storage 2 - Indicates the amount of memory, assigned to a user job, that may be shared with General Loader during loading. If all of the memory requested in storage 1 will have data or other information loaded into it by General Loader, then storage 2 will be omitted. Storage 2 will not be used with JOVIAL.

Print Lines - Specify the maximum number of lines to be written on SYSOUT during program execution for later printing.

1. The $ LIMITS control card should follow the $ EXECUTE control card.

2. If no $ LIMITS control card is used for an execution, a standard set is automatically provided. Standard limits for execution are:

3 minutes	(.05 hundredths)
16k	16 x 1024 words
0	No Overlay
5000	Lines to SYSOUT

3. It is sometimes necessary, for extremely large compilations, to increase the standard limit provided for compiling JOVIAL programs. To do this, a $ LIMITS control card is placed immediately after the $ JOVIAL control card, ahead of the JOVIAL source program. It must be remembered, this $ LIMITS control card has nothing to do with the execution.

Standard compilation limits are:

.08	hundredths
28k	28 x 1024 words
0	No overlay
10000	Lines to SYSOUT

$ ENDJOB

1	8	16
$	ENDJOB	Not Used

The $ ENDJOB control card is used to indicate to System Input that the job being processed is a candidate for allocation and execution, provided errors were not detected. If errors are detected, they are noted on the console typewriter as they occur, System Input completes the processing, and the entire job is deleted without being allocated.

The $ ENDJOB control card is the last card of every job.

***EOF

1	8	16
***EOF	Not Used	Not Used

The ***EOF control card signals the card reader that an EOF status has occurred and the card reader is released.

APPENDIX A

JOVIAL COMPILATION ERROR MESSAGES

The following list contains all current JOVIAL compilation error messages, their meanings, and possible remedial action.

Error Message	Meaning and Action
ACTUAL-FORMAT PARAM LISTS DO NOT AGREE	The procedure or function call argument list being processed does not agree in number with the formal argument list description. The statement is not compiled.
ARRAY EXCEEDS REASONABLE SIZE	The space required for ARRAY is greater than 24,000. No space is allocated.
BAD STATUS RELATIONSHIP	The status constant being compared to this status variable is not one of the previously declared legal values. The statement is not compiled.
BAD XEC NAME	The Terminal Action parameter is not a procedure call or a zero. The statement is not compiled.
CIRCULAR DEF	The expression being compiled contains a redefinition (by the DEFINE declaration) which is improperly constructed. The statement using this definition is not compiled.
COMPILER DUMMY BUFFER HAS OVERFLOWED	The compiler has exceeded the capacity of the internal dummy buffer in this compile process.
COMPILER EXIT BUFFER HAS OVERFLOWED	The compiler has used up the allocated buffer area for its recursive calls. Compilation is terminated.
COMPILER WORK BUFFER HAS OVERFLOWED	The compiler has exceeded the capacity of the internal work buffer in this compile process.
COMPOOL FILE NOT PRESENT	No COMPOOL (.L) file has been allocated to this activity but a "COMPOOL" list has been encountered.
COMPOOL FORMAT ERROR	An error in the COMPOOL (.L) file has been detected while searching/processing an entry.
CONFLICTING USE	The variable being compiled is being used in an improper context. The statement is not compiled.

CONST OVFLO	The exponent part of the constant being compiled is larger than 2^{127}. The exponent is set to zero.
FIELD TOO BIG	An item has been declared too large for its type. The declaration is not compiled.
FILE NAME REQUIRED	An I/O request has been found without a file name in the parameter reserved for it. The I/O statement is not compiled.
FLOATING ITEM REQUIRED	A floating point item is expected within a floating point function. The statement is not compiled.
ILLEGAL CHARACTER	A character input from the source deck is illegal. It is ignored.
ILLEGAL CONSEQ STATEMENTS	An "IF" or "IFEITH" statement is followed immediately by a "FOR", "IF" or "IFEITH" statement; or, a "FOR" statement is followed immediately by an "IF" or "IFEITH" statement. BEGIN-END construction should be used to separate the consecutive statements.
ILLEGAL CONVERSION	In the assignment or exchange statement being compiled, the data types are incompatible; that is, no implied conversion is possible. The statement is not compiled.
ILLEGAL DATA LENGTH PARAMETER	The parameter defining the length of the data in the I/O transfer is illegal. The statement is not compiled.
ILLEGAL FOR COMPOUND	An IF statement followed immediately by an END, which is not associated with a FOR, is an illegal construction. The IF statement is not compiled.
ILLEGAL PRIMITIVE	A name beginning with a prime character has been encountered, but it is not among the allowable primitives. The statement is not compiled.
ILLEGAL START BIT	An illegal attempt has been made to define a literal field as starting at other than a byte boundary (multiple of 6-bits).
ILLEGAL STATUS ASSIGNMENT	The status constant being assigned to this status variable is not one of the previously defined legal values. The statement is not compiled.
JUNK PROGRAM	A combination of errors so extensive that the compilation cannot continue has been detected; compilation terminates.
LOOP CONTROL	Multiple three-factor FOR statements are used in parallel. Erroneous code may be generated.
MAX SYMBOL LENGTH EXCEEDED	A symbol exceeds 143 characters in length.
MISSING ASTERISK	The asterisk (*) is omitted from the exponentiation brackets. It is assumed.
MISSING BEGIN	There is an absence of a BEGIN in a PROG declaration. The BEGIN is assumed.

MISSING DOLLAR	The dollar sign ($) is missing from a subscript bracket or a statement terminator. It is assumed.
MISSING END	In processing a statement in which an END is required, the compiler has encountered a symbol that is not syntactically correct and that is not an END. Or, the end of a program was reached with a count of begins greater than the count of ends. The effect on compilation depends on the particular occurrence.
MISSING SLASH	The slash (/) is omitted from the absolute value brackets. It is assumed.
MISSING START	The START card does not appear first in the JOVIAL program deck. It is assumed.
MISSING TERM	An end-of-file condition has been encountered on the source (S*) file and no TERM $ statement has been found.
MUL DEF	The variable name currently being used has been defined more than once in this program. Erroneous references to this variable result.
MULTI STA ERROR	This is a combination of errors in continuous statements. The statements are not compiled.
MULTIPLE UNARIES	Multiple arithmetic or Boolean unary operators (-, +, NOT) occur in an expression. The expression is evaluated as it stands.
NAME NOT IN COMPOOL	A COMPOOL list contains a name that is not in COMPOOL. The name is ignored.
NESTED PROC	The PROC statement being compiled is in the scope of another procedure declaration. The PROC statement is not compiled.
NO PATTERN TABLE	No pattern table was declared for this table, which was declared as LIKE. The LIKE declaration is not compiled.
NO PROC FOR RETURN	A RETURN statement appears outside the scope of a procedure declaration. The RETURN statement is not compiled.
NOT IMPLEMENTED, PLEASE TRY DIFFERENT CONSTRUCTION	A compiler error is encountered which is probably caused by an erroneous construction in the indicated statement. If construction is altered and compiler error persists, send dump to HISI, PCO, Phoenix.
NOT TABLE ITEM	A name is subscripted that is not declared in a table. The statement is not compiled.
NUMERIC TYPE REQUIRED	A numeric argument is required during processing of an absolute function. The statement is not compiled.

OVERLAY NAME ILLEGAL	A name used in an overlay statement is not allowed; or, a numeric has been encountered in a subordinate overlay statement.
PARAMETER NOT PERMITTED IN DIRECT CODE	The arguments of a called procedure are referenced within direct code. The statement is not compiled.
PACKING REQUIRED	A packing parameter is missing in the declaration of a programmer specified table item.
PRECLUDED BY CONTEXT	A comparison of the usage of this variable or constant and its type indicates an inconsistency. The statement is not compiled.
PRESET CONST ERROR	Something other than a legal constant is in PRESET LIST or there is a missing END. Remaining preset constants are invalid.
RIGID TABLE - ILLEGAL TO SET NENT	The NENT of a rigid table is preset and cannot be changed by the programmer. The statement is not compiled.
SINGLE PRECISION MULT OVERFLOW, SIGNIFICANT BITS LOST	During the calculation of the indicated statement consisting of single precision constants, multiplication operation has returned a double precision result. Since single precision is indicated by the constant expression, the 36 least significant bits of the double precision multiplication result will be saved and the most significant bits will be lost. The compiler will then continue processing the indicated statement.
SUBSCRIPT MULDEF	The "FOR" loop variable being declared is already defined in an open "FOR" loop.
SUBSCRIPT UNDEFINED	The "FOR" loop variable used as a subscript does not belong to an open "FOR" loop.
SUPERFLUOUS END	There is an extra END or missing BEGIN in the program. This can be an indication of possible serious trouble. The effect on the compilation depends on the particular occurrence.
SYNTAX	An error is detected in the construction of the statement currently being compiled. The statement is not compiled.
TABLE ITEM NAMES NOT PERMITTED IN DIRECT CODE	A table item has been used in direct code. This statement is not compiled.
TWO FEW ARGS	The BIT or BYTE modifier being compiled does not have any defining parameters. The statement is not compiled.
TOO FEW ENDS	END bracket(s) not found before end of program. Compile terminates normally but results may not be correct.

TOO FEW SUBSCRIPTS	The subscripted item being compiled has fewer subscripts than were contained in its original definition. The statement is not compiled.
TOO MANY ARGS	The BIT or BYTE modifier being compiled has more than two defining parameters. The statement is not compiled.
TOO MANY SUBSCRIPTS	The subscripted item being compiled has more subscripts than were contained in its original definition. The statement is not compiled.
UNDEF	The statement name currently being used has not been defined in this program. Erroneous transfers may result.
WARNING-RECEIVING ARG SIZE OF ZERO POSSIBLE FOR SPECIAL VARIABLE	A JOVIAL special variable has been used in such a way in the indicated statement that the receiving argument for the data returned by the special variable has an implied size of zero words. The compiler continues execution, attempting to generate code for the indicated statement. Generated code may be bad.
WARNING-THIS STA POSSIBLY NON-ACCESSIBLE	Program statement is not accessible; statement is flagged on compiled listing.
$ IN COMMENT	A $ sign was encountered while processing a comment. Everything up to the $ is considered a comment, and the symbol after the $ is considered the beginning of a new statement.
XXXXX WORDS ADDITIONAL CORE ALLOCATED FOR THIS COMPILATION	If more core is required for compilation than is allocated, the compiler will request additional core in 1024-word blocks. If requests are satisfied as additional core is needed, compilation will continue and this message will appear at the end of compilation, advising the programmer of the additional amount of memory which was obtained during the entire compilation process.

APPENDIX B

MACHINE LANGUAGE (GMAP) ERROR FLAGS

A list of error flags, which may appear on the left margin of an object code listout, are as follows:

Flag	Meaning
U	Undefined symbol(s) appear in the variable field.
M	Multiple defined symbol(s) appear in the location field and/or the variable field.
A	Address error. Illegal value or symbol appears in the variable field. Also used to denote lack of a required field.
X	Illegal index or address modification.
R	Relocation error. Expression in the variable field will produce a relocatable error upon loading.
P	Phase error. This implies undetected machine error or symbols becoming defined in pass two with a different value from pass one.
E	(1) For DIRECT code, flags an illegal character in the E/O field. (2) For JOVIAL code, flags a statement which did not compile and for which a MME GEBORT is inserted.
C	Error conversion of either a literal constant or a subfield of a data-generative pseudo-operation. Illegal character.
L	Illegal operation.
T	An assembly table overflowed not permitting proper processing of this card completely. Table overflow error information will appear at the end of testing.

APPENDIX C

ENCODING OF SIGNS

Hollerith Encoding

First Octal Digit \ Second Octal Digit	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	[	#	@	:	>	?
2	␣(1)	A	B	C	D	E	F	G
3	H	I	&	.	]	(	<	\
4	↑	J	K	L	M	N	O	P
5	Q	R	-	$	*	)	;	'
6	+	/	S	T	U	V	W	X
7	Y	Z	←	,	%	=	"	!

Transmission Encoding

First Octal Digit \ Second Octal Digit	0	1	2	3	4	5	6	7
0	%	↑	↓	[	]	␣(1)	A	B
1	C	D	E	F	G	H	I	J
2	K	L	M	N	O	P	Q	R
3	S	T	U	V	W	X	Y	Z
4	)	-	+	<	=	>	≤	$
5	*	(	≥	:	]	∧	,	≠
6	0	1	2	3	4	5	6	7
7	8	9	∨	;	/	.	→	=

Note: (1) ␣ indicates a blank space.

APPENDIX D

JOVIAL LANGUAGE RESTRICTIONS

The following are language restrictions in JOVIAL:

- DUAL items are excluded.
- Medium table packing is treated like dense table packing. *
- The J3X I/O is implemented.
- GMAP assembly code (excluding all macro instructions) is the only legal type of direct code.
- To conform with the conventions of the operating system, a $ must not appear in column 1 of an input card.
- To conform with the conventions of the loader, procedure names for which SYMREFs must be generated must not exceed six characters. *

APPENDIX E

JOVIAL USE MEMOS

This appendix contains a series of memos issued in response to JOVIAL user problems. The intent is to update the appendix each time a new memo is issued.

USE MEMO 1

JOVIAL Source Passed Through CARDIN

Long literal items must be restricted to the space available on a single line of TTY input.

If a user continues his long literal item to the next line he will end up with a minimum of twelve embedded blanks in the long literal item thus abrogating the item declaration. This will result in a syntax error, multistation error, and a FO abort caused by the implied bounds of the improperly expanded long literal item.

USE MEMO 2

Use Memo 2 is no longer applicable.

USE MEMO 3

Justification of Preset Data (Literals)

Simple Item - Preset data is right-justified within the declared field. Any unused byte positions in the field and/or the compiler-allocated words are "filled" with blank characters.

Ordinary Table Item -

1. Data is right-justified when the preset has fewer bytes than the declared field. Any unused bytes are "filled" with blank characters.

2. The declared field is right-justified in the compiler-allocated field. (The declared field is a given number of bytes but the compiler allocates the minimum number of full words required.) Any unused bytes (always preceding the preset) are "filled" with blank characters.

Users are cautioned to use care in declaring and presetting items and tables to ensure the proper positioning for later access of data. Data stored into a declared field through use of the IOn statement is done only in full word multiples and storage is begun at the location specified with the dn parameter of the IOn statement. (See Section IX - I/O Operations)

USE MEMO 4

Calls to External Subroutines

JOVIAL allows symbols to be of n length; however, GCOS constrains symbols to a maximum of six characters.

Programmers are advised to utilize a maximum of six characters for any JOVIAL symbol that is external to the user program. Violation of this rule, as in the case of a missing external routine, will cause a TSX7 to a MME GEBORT to be inserted as the first executable statement of the JOVIAL program to be executed. The TSX7 is inserted by the Loader and is not a compiler malfunction.

USE MEMO 5

JOVIAL Binary on B*

To record the JOVIAL object (B*) file on magnetic tape the programmer must ensure that the activity, in which he has declared B* as magnetic tape, has an $ EXECUTE card as one of the control cards.

If the $ EXECUTE card is omitted, the B* file will not be written.

USE MEMO 6

Maximum Length of Long Literal Items

A long literal item is defined as being more than six Hollerith characters. The maximum length of a long literal item is twenty words or 120 Hollerith characters.

Longer strings of data can be encoded through the use of concatenated table items within a table entry.

USE MEMO 7

IOn Statement Restrictions

Use of an IOn statement is restricted to the actual transfer of data, one or more words. The user may not use zero as the number of words to be transmitted. The run time routine (JVLA) will indicate a buffer size error and the I/O will be ignored.

USE MEMO 8

.FEXIT - Program Abort

A $ USE .FEXIT/1/ card may be used with a JOVIAL program to cause an F7-abort to occur after program execution. It provides a method of obtaining a core dump without inclusion of a MME GEBORT within the program when the program would otherwise terminate normally.

The $ USE card is formatted as follows:

```
1       8          16
$       USE        .FEXIT/1/
```

SAMPLE DECK SETUPS:

a. Compile and Execute

```
$  SNUMB
$  IDENT
$  OPTION JOVIAL
$  USE .FEXIT/1/
$  JOVIAL
$  INCODE IBMF
   JOVIAL SOURCE DECK
$  EXECUTE DUMP
   DATA (if any)
$  ENDJOB
```

b. Execute

```
$  SNUMB
$  IDENT
$  OPTION JOVIAL
$  USE .FEXIT/1/
$  OBJECT
   OBJECT DECK
$  DKEND
$  EXECUTE DUMP
   DATA (if any)
$  ENDJOB
```

USE MEMO 9

Maximum Amount of Preset Allowed for Table Items

TABLE item presetting is limited to 1024 words. Exceeding the limit will recycle the table pointers to zero thus causing the loss of original preset data. These restrictions apply to any single table item although it may occur within many entries of the table.

(TABLE ITEM PRESET WORDS) (NUMBER OF ENTRIES) = WORDS OF Preset Data. (May not exceed 1024.)

USE MEMO 10

Use Memo 10 is no longer applicable.

USE MEMO 11

Use Memo 11 is no longer applicable.

APPENDIX F

TIME-SHARING JOVIAL

TIME-SHARING JOVIAL SUBSYSTEM

The time-sharing JOVIAL subsystem provides full time-sharing capabilities with the JOVIAL language processor. The subsystem provides the time-sharing user with all conventional time-sharing commands. The commands operate in the JOVIAL environment in a manner compatible with other time-sharing language processor subsystems.

A mechanism has been provided within the time-sharing JOVIAL subsystem which utilizes the batch JOVIAL compiler to perform time-sharing compilations and perform a "pre-load" of the compilation results, resulting in a directly loadable file (H*) which is then loaded and executed under the time-sharing executive. The compilation and load process is a single activity which involves compilation and loading when a source program is submitted and a load-alone if a binary file(s) is submitted by means of the RUN command.

Commands available under the time-sharing JOVIAL subsystem permit generation, modification, and utilization of JOVIAL programs and data files. Other commands involve program compilation-execution requests. The following commands are available:

ABC	DONE	LIB	REMOVE
ACCESS	EDIT	LIST	RESAVE
ASCASC	ERACE	LUCID	RESEQUENCE
ASCBCD	FDUMP	NEW	ROLLBACK
AUTOMATIC	GET	NEWUSER	RUN
BCDASC	HELP	NOPARITY	SAVE
BPUNCH	HOLD	OLD	SCAN
BPRINT	JABT	PARITY	SEND
BYE	JOUT	PERM	STATUS
CATALOG	JSTS	PRINT	SYSTEM
DELETE	LENGTH	PURGE	TAPE
		RELEASE	

An explanation of each of these commands may be found in Time-Sharing System General Information Manual.

RUN COMMAND

The RUN command as used with time-sharing JOVIAL is different from most other subsystem RUN commands in that it is a superset of these commands. The RUN command in the time-sharing JOVIAL subsystem has the following format and meaning:

RUN [H][-nnn] fs = fh;fc (opts) flib # fe

where braces indicate optional fields and the following apply:

H causes the standard time-sharing header to be printed as the job begins to run. It is optional and, if omitted, the header will not appear.

-nnn allows the user to specify the number of seconds he wishes to limit the executing object program (not the compile and load activity) to run. Execution time will be set to nnn seconds, where nnn is less than or equal to 180. If omitted, unlimited program execution time is allowed. Execution may be halted with the BREAK key.

fs is a set of file descriptors for input to the compiler and/or loader. The set may consist of source codes as BCD card image files (media code 3), time-sharing format source files (media code 6), or binary card image files (media code 1), or may consist of a single file descriptor which points to a previously generated system loadable file (H*). This field is optional, and if missing, indicates that the current file (*SRC) is to be operated upon. If more than one file descriptor is desired (i.e., a compile and necessary binary files of called subroutines or a series of binary files to be executed), multiple descriptors are semicolon-separated.

= must be included whenever the fh, fc, (opts), or flib options are desired.

fh is a single file descriptor pointing to a random file into which the system-loadable file (H*) produced by the loader will be saved if the compilation is successful. If this field is absent, the H* file is generated into a temporary file and released prior to the program just compiled being placed into execution under the time-sharing executive. The presence of this option is valid only when the program indicated by the field fs, the time-sharing library, and the user libraries (if any) are bindable (i.e., there are no outstanding SYMREFs). If outstanding SYMREFs exist, the H* file will be created and executable but any reference to an unsatisfied SYMREF will cause an abort (the loader inserts a MME GEBORT at unsatisfied references). The time-sharing executive then simulates an illegal operation fault.

; must be included even when the fh option is omitted.

fc is a single file descriptor which points to a sequential file into which the JOVIAL compiler is to place the binary (*C) result of an indicated compilation. In this case, the field fs plus the libraries need not indicate a complete program; i.e., individual or collections of subroutines may be compiled and saved. This field is optional and if missing, a C* file will not be generated.

(opts) is a set of options, contained within parentheses and comma-separated, which are listed below. When omitted, the underlined options are assumed.

LNO - the source input records are line-numbered beginning in column 1 and terminating with the first nonnumeric character.

NLNO - the JOVIAL source records are not line numbered.

GO - the program will be executed at the completion of compilation. The object program will be saved if specified.

NOGO - the program will not be executed at the completion of the compilation. If specified, the object program will be saved (H*) but if not specified, only compilation will be done.

ULIB - File descriptors exist following the end of the options field which locate user libraries. These libraries are to be searched for missing routines prior to sending for them in the system library.

NULIB - No user libraries are to be searched.

ASCII - character data to be utilized by the object program which is a result of this compile will be encoded in the 9-bit ASCII character set; i.e., a time-sharing file or direct terminal I/O.

BCD - character data to be utilized by the object program as a result of this compile at execution time will be encoded in 6-bit BCD standard character set; i.e., a BCD file.

TIME = nnn - The batch compilation and/or the loader activity time limit will be set to nnn seconds, where nnn is less than or equal to 180. If not specified, nnn will be set to 180.

URGC = nn - The batch compilation and/or the loader urgency will be set to nn, where nn is less than or equal to 40. If not specified, nn will be set to 40.

CORE = nn - The batch compilation core requirement will be set to (1) 23k,nn 22; (2) nn + 6k, 22 nn 28; and (3) 23k,nn 28. The loader core requirement will be set to nn + 6k, where nn should reflect the size of the object program with the needed library routines. The 6k reflects the space needed for the loader. If not specified, nn will be set to 16.

This option is most useful in limiting or expanding the amount of core allocated for the loader activity which follows compilation and prepares the object program for time-sharing execution. As noted, it is also useful if 28k is not large enough.an allocation for the compilation, in which case the allocation can be expanded to 34k.

REMO - Temporary files created for the batch process will be removed from the AFT (Available File Table) after the termination of the activity.

TEST - Test version of the JOVIAL compiler and/or the loader is to be used for the batch activity. A file of the name JOVIAL49 (which contains the test compiler) must be an accessed file (in the AFT) before this argument is used in the RUN command. If these two conditions are met, then the file JOVIAL49 will be allocated as file code ** in the batch activity.

NAME = - This provides a name for the main link of the saved H* file. It may be used both at the time of creation of this file and subsequently as it is reused. The name is placed in the SAVE/ field of the $ OPTION card.

flib is a sequence of file descriptors pointing to random files containing user libraries to be searched before the system library. These user libraries must previously have been edited on the file as random system library files by the object library editor. This field is optional but must be present when the ULIB option is specified.

\# must be included whenever the fe option is desired.

fe is a set of file descriptors for files which will be required during execution; i.e., I/O files. Each descriptor must specify a file name (or an alternate name, if necessary) of the form nn, where nn ranges between 01 and 44 inclusive. This name, or alternate name, represents a logical file code referenced by I/O statements in the program. I/O may be terminal-directed by specifying a descriptor in the form "nn". File codes 05, 06, and 41-43 are implicitly defined for terminal I/O and need not be mentioned in the RUN command unless I/O through these file codes is to be redirected to a file (see "Terminal I/O" below).

USAGE

For batch JOVIAL users, the BCD character set has been and is the way of life. In the time-sharing environment, ASCII is the current character set. In designing and implementing the time-sharing JOVIAL subsystem, the need has been to approach both worlds. Since both approaches have their particular advantages, the user may now submit either ASCII or BCD source statements to the JOVIAL compiler from either the time-sharing or batch environments. Before compile time, the user is not required to specify the type of source to be submitted to the compiler but he is required to specify the desired orientation (ASCII or BCD) for the object program which will result from the compile. If not specified, the default in batch is BCD, and in time-sharing is ASCII.

From the time-sharing JOVIAL subsystem, the user may enter a program from the terminal and RUN it or he may RUN a program which has previously been saved on a file. The saved program may be either ASCII- or BCD-type file. From the batch environment, the user may submit a program in any of the various methods he now has available, including programs saved on files. These saved programs may be of either ASCII or BCD form.

ASCII or BCD source files (programs saved on perm-files) supplied to the compiler in the batch environment are assumed to have no line numbers preceding the source statements. If one wishes to compile in batch a source file with line numbers (i.e., one created at the terminal), an additional control card must be added to the deck setup within the compile activity. The additional card is the $ SET control card and it must be used to set bit 15 of the switch word before the compiler sees the source file. The format of the $ SET control card is as follows:

1	8	16
$	SET	15

When running in the time-sharing JOVIAL environment, the object module will normally have ASCII properties; i.e., it will accept and output data in ASCII form only. If the user wishes the module to have BCD properties (i.e., accept and/or output BCD characters), he has only to specify the BCD option on the RUN command. When binding a JOVIAL program and various externally compiled procedures, the modules may be of mixed modes; i.e., modules may have ASCII or BCD or various combinations of properties (the only restriction is that a particular module exhibit only one characteristic). They will run successfully together, processing and outputting whatever mode designed for and communicating with one another as procedures and main program normally communicate.

Note

All object modules, whether exhibiting ASCII or BCD characteristics, work internally with the six-bit BCD byte rather than the nine-bit ASCII byte. Therefore, it is illegal to program in the JOVIAL source language for the nine-bit ASCII byte when ASCII characteristics are desired; i.e., program as usual with Hollerith characters. This accounts for the reason why executing modules exhibiting different characteristics can still communicate with one another. The characteristics exhibited relate to the specific I/O routine library routines which are called from within the compiled module(s).

When compiling in the batch environment, the object module will normally have BCD properties; i.e., it will accept and/or output data in BCD form only. If in the batch environment, the user desires the batch compiled object module or modules to exhibit ASCII characteristics while executing, he is required to add one additional control card, $ EQUATE, to the deck setup just prior to the $ EXECUTE card. The required format of the $ EQUATE card is

```
1       8       16
$       EQUATE    .VIOT/.VIO/
```

Unlike the time-sharing environment, when binding together a main program and procedures which have been compiled in batch for execution, the modules may not exhibit mixed modes but must all be either BCD by default or ASCII by choice. But the user may obtain the same ASCII-BCD mixed-mode data capabilites in the batch environment at execution time as one has in the time-sharing environment. At execution time, the user binds together object decks compiled and saved in the time-sharing environment and exhibiting ASCII characteristics with object decks compiled and saved either in time-sharing or batch environments and exhibiting BCD characteristics. Such a procedure will allow the executing program(s) to access both BCD and ASCII data.

BCD input/output capabilities in the time-sharing environment extend to file and terminal I/O (i.e., routines with BCD characteristics can request from or send to the terminal any desired information as well as writing and/or reading file in a BCD format), but ASCII input/output capabilities in the batch environment are restricted to file I/O. For maximum utility of both BCD and ASCII files, a variety of on-line conversion routines exist in the time-sharing JOVIAL subsystem for converting files between ASCII and BCD. All the above capabilities allow the user to compile and save C* files in both the batch and time-sharing environments in either mode and later execute them in either environment and be able to input properly to the modules and examine output. Since there is no ASCII System Input or ASCII SYSOUT in batch, one cannot do ASCII I/O on standard system I/O devices in that environment. Thus files 05, 06, 41, 42, and 43 cannot be linked to ASCII I/O routines in batch but a link can be made to files -- tape, disc, and drum.

With the inception of the time-sharing JOVIAL subsystem, ASCII files in standard system format will appear. With the advent of the YFORTRAN system, a file format for ASCII information which conforms in all ways to I/O rules has been defined. The new standard system format ASCII file is the format expected in the ASCII environment. Data files in ASCII, and only data files (not source files to be passed to the compiler for compilation), must be converted from the regular time-sharing format in which they were created at the terminal to the new format before being utilized as data files for an object JOVIAL program. This may be accomplished by time-sharing command, ASCASC. The command converts time-sharing ASCII format files to standard system format files and new standard system format files (back) to time-sharing format files, based on the mode of the input file to the command. The command is similar in all respects to the BCDASC and ASCBCD commands, including all features. If ASCII is specified for the object JOVIAL program mode, all output as well will be ASCII files in the new standard system format unless output is directed to the terminal (in which case the File and Record Control facility handles it directly). If the user at a terminal wishes to examine an ASCII file written by an object JOVIAL program, he must first convert it to the time-sharing format with the ASCASC command, after which time he may examine the file at the terminal. However, if he wishes to use it directly as data to another ASCII mode program, he need not do any conversion because it was created in a proper format for direct usage by JOVIAL ASCII modules.

Note

Direct terminal input does not have to be converted if data is entered in direct response to the data call equals sign, (=); the File and Record Control facility handles the data directly.

FILE CHARACTERISTICS

Compile - Batch

BCD Files - Standard System Format (media code 3)

ASCII Files - Time-sharing Format (media code 5)

Compile - Time-sharing

BCD Files - Standard System Format (media code 3)

ASCII Files - Time-sharing Format (media code 5)

Batch I/O (Object Program)

BCD Files - Standard System Format (media code 3)

ASCII Files - Standard System Format (media code 6)

Binary Files - Standard System Format (media code 1)

Time-sharing I/O (Object Program)

BCD Files - Standard System Format (media code 3)

ASCII Files - Standard System Format (media code 6)

Binary Files - Standard System Format (media code 1)

RUN EXAMPLES

1. RUN

   Current *SRC JOVIAL source file will be compiled and executed.

2. RUNH-20 JV001=HSTAR;CSTAR1(ULIB)BCDIO;
   ASCIO#INPUT"01";OUTPUT"02"

   JOVIAL program file JV001 is to be compiled and executed. The H* will be saved on file HSTAR and the C* on file CSTAR1. For the execution, the user libraries BCDIO and ASCIO will be scanned for outstanding SYMREFs in JV001. ASCIO contains routines which perform ASCII I/O and BCDIO contains routines which perform I/O through logical file codes 01 and 02 which have been given alternate names here, in the AFT, of INPUT and OUTPUT (files which actually exist). A header will precede the execution report and the object program will be limited to 20 seconds of execution time.

3. RUN # "10"

   Current *SRC file will be compiled and executed and input or output directed through logical file code 10 will be directed to/from the terminal.

4. RUN # INPUT "05"

   Current *SRC file will be compiled and executed and input normally requested from the terminal (file code 05) will be channelled from file INPUT instead.

5. RUN JV002=HSTAR(NOGO,NLNO)

   JOVIAL file JV002 will be compiled and the H* saved on the file HSTAR. File JV002 has no source statement line numbers.

6. RUN BCDIOM=;CSTAR2(BCD,NOGO)

   JOVIAL file BCDIOM will be compiled and the C* will be saved on the file CSTAR2. It is desired that the object program have BCD capabilities. The compiled program will not be executed and the H* will not be saved.

7. RUN HSTAR

   Execute a previously bound and saved H* file. The HSTAR file has no outstanding SYMREF and does only terminal I/O.

8. RUN =(NLNO)#TAP"01";DISC"02";DRUM"03"

   Compile and execute the current *SRC file which has three logical file codes: 01,02, and 03. These file codes are equated to actual files TAP, DISC, and DRUM which exist and have or will accept data.

9. RUN WRKBCD=(BCD)#INPUT"05";OUTPUT"06"

   Compile and execute the JOVIAL program WRKBCD and desire the object program to have BCD capabilities. WR KBCD has input (05) and output (06) normally directed to the terminal but instead are here redirected to the BCD files INPUT and OUTPUT.

10. RUN-60 =HSTAR(TIME=60,CORE=16,URGC=05,ULIB)SEARCH

    Compile and executed the current *SRC file. Limit the compile time to 60 seconds, the core size to 16k, and urgency to 5. This limits the object program execution time to 60 seconds. Finally, the user library SEARCH is to be searched for outstanding SYMREF in the *SRC program. The bound H* file is to be saved.

11. RUN STEP;CSTAR1;CSTAR2

    Compile and execute the JOVIAL program file STEP and bind it with two previously saved C* files: CSTAR1 and CSTAR2.

12. RUN *;CSTAR1;CSTAR2

    Compile and execute the current *SRC file and bind it with two previously saved C* files: CSTAR1 and CSTAR2.

Note

As in many of the examples above where files are referenced in the RUN command, users should remember to access these files (put them in the AFT) before typing the RUN command. I/O requests to unaccessed files trigger time-sharing GFRC to output the message:

FILE fc NOT IN AFT. ACCESS CALLED

"fc" is the referenced file code. The time-sharing subsystem ACCESS is then called to query the user. The execution continues when the user exits from ACCESS. Prior accessing of files creates better RUN efficiency.

ACCESSING THE TIME-SHARING JOVIAL SUBSYSTEM

The time-sharing JOVIAL subsystem is a system-level language processor. Therefore, it can be entered at the system level by responding to the question as below:

```
SYSTEM? JOVIAL
```

Once having responded this way, all previously described commands are available in the subsystem, including the special RUN command. JOVIAL programs can be built and then compiled and executed once in this subsystem. It must be noted, however, that JOVIAL programs can be compiled and executed only with the RUN command peculiar to the subsystem JOVIAL (except for the terminal-batch interface through CARDIN). RUN commands within other language subsystems are not designed to accommodate JOVIAL compiles and executions.

TERMINAL I/O

The object program, the result of a JOVIAL compile, is placed in execution under the time-sharing executive by the RUN subsystem. Once in execution, the user can dynamically interact with the program in the time-sharing JOVIAL subsystem. Communication is accomplished through time-sharing I/O -- a special version of File and Record Control routines designed to allow dynamic interaction through the terminal. The next few paragraphs are intended as an introduction to means of interfacing with the executing object program through time-sharing I/O.

Time-sharing I/O has several special file codes which are implicitly defined for terminal I/O unless purposely directed elsewhere. They are:

```
"05"  terminal input
"06"  terminal output
"41"  terminal input
"42"  terminal output
"43"  terminal punch.
```

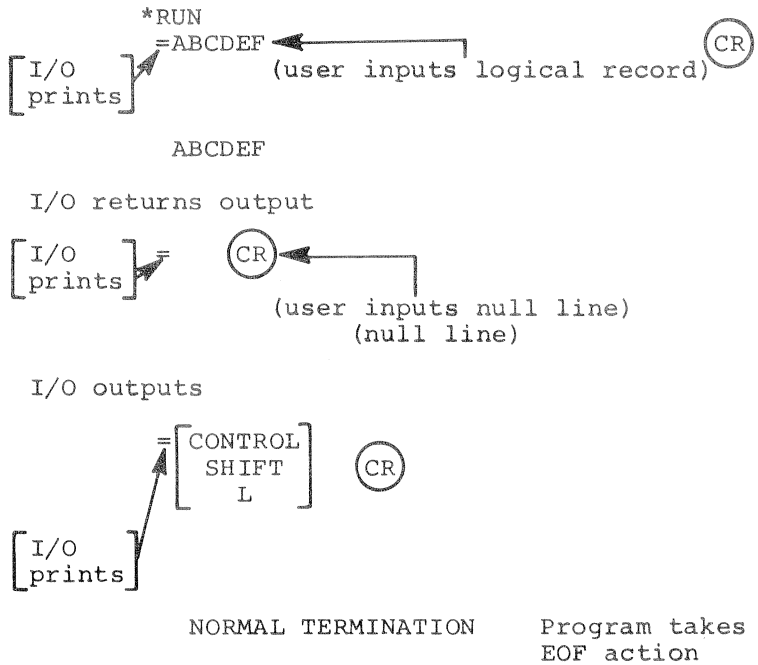
As seen earlier in a RUN example, other file codes may be forced to the terminal for I/O although not implicitly defined as such.

When, in a JOVIAL program, the user requests terminal I/O, time-sharing I/O will reflexively read input from the terminal. The equals sign will be issued before the user inputs his response to satisfy the request. A carriage return response only sends null input (blanks) back to the object JOVIAL program. A carriage return following data signals the end of the current record of input. The special character "CONTROL-SHIFT-L" followed by a carriage return as response to an equals sign signifies an end-of-file on input.

For purposes of example, terminal I/O may be thought of as card I/O. There are a maximum of 72 characters which may be requested per line and a maximum of 72 characters which may be output per line. Assume the object program of the following source executing:

```
PROGRM VIOTST
  START
    ITEM BUFF H 72 $
    FILE READ V(OK) V(NULL) R05 $
    FILE PRNT V(OK) V(NULL) R06 $
    IN(1,READ) $ OUT(1,PRNT) $
    XX. IO Ø1(0,READ,0,BUFF,12) $
    IF READ EQ V(EOF)$ GOTO YY $
       IO Ø2(3,PRNT,0,BUFF,12) $  GOTO XX $
    YY. OUT(2,PRNT) $ IN(2,READ) $
    TERM $
```

Terminal interaction through time-sharing I/O with this program while executing might look like the following:



PAPER TAPE I/O IN TIME-SHARING JOVIAL

This option applies to both ASCII and BCD capabilities. On all read statements issued to the terminal, paper tape input may be substituted.

Output directed to file code 43 (punch file) and directed to the terminal can be punched on paper tape. The user should prepare a leader on the tape and turn the paper tape punch on. The punch routine will properly format the tape from that point on.

To accomplish data input from paper tape, the tape lever must be set to the start position. Data on the tape should adhere to the rules for entering data from the terminal. A carriage return, LINE-FEED, RUBOUT must follow each line of input data.

Restrictions

1. Attempts to direct binary I/O to the terminal will result in an abort with an appropriate message.
2. Attempts to REWIND or write an end-of-file on file codes 05, 06, 41, 42, 43 will cause an abort with an error message.

Time-Sharing Abort Messages

1.	GET CODE 5 - fc	Variable record size is zero.
2.	PUT CODE 4 - fc	Current logical record larger than buffer. (JOVIAL, though, will partition when it can).
3.	TELETYPE O/P NOT ASCII ON - fc	File media code not 6.
4.	CLOSE CODE 3 - fc	File to be closed is not in logical unit table chain.
5.	FILE SPACE EXHAUSTED - fc	Attempts to grow the file being written have failed.
6.	PUTSZ CODE 2 - fc	New size larger than available buffer space.
7.	BACKSPACE ERROR - fc	A bad status has been returned after attempt to backspace, or, file is random, or file is not open, or file has no EOF.
8.	READ ERROR - fc	Bad status returned after DRL to read file.
9.	WRITE ERROR - fc	Bad status returned after DRL to write to a file.

FAULTS IN TIME-SHARING EXECUTION

Overflow and divide checks are processed by the library fault processor.

Op code, memory, and tag faults are trapped and one of the following messages printed before the abort:

OP / MEMORY / TAG FAULT

The location of the fault may be obtained by routing the core dump file to an "ABRT" file and snapping the fault vector locations from this file. See Time-Sharing System-Programmer's Reference manual, on DRL ABORT.

MME faults are detected and the message is printed:

MME FAULT

INTERPRETING ERROR DIAGNOSTICS

Special mention must be made of error diagnostics to avoid what might be a potential problem for the time-sharing JOVIAL user. After the compilation of a JOVIAL program, any source errors recognized by the compiler in the program are returned to the user and listed at the terminal before the program is placed in execution or the compile and load activity is terminated. Errors output from the compiler are displayed in the following formats:

ERROR....STA XXXX Diagnostic of usage error

ERROR..STA..XXXX Diagnostic of syntax error

The diagnostic informs the user that an error occurred at a specified statement number in the JOVIAL program and that the error is of the type described in the message. Refer to Appendix A for a list of compilation diagnostics (error messages).

The statement number given in the error diagnostic is not necessarily a source statement line number which usually begins each source code line created at the terminal. The statement number instead refers to a specified JOVIAL source statement. It is, at times, likely that a JOVIAL source statement may not fit on one line; it can therefore be extended to the next line(s), or the statement may, at will, be spread out over more than one line. It is also possible that more than one JOVIAL source statement may appear on one given line. For these reasons, it is obvious that the source code line number is not necessarily the same as the JOVIAL statement number. Further, JOVIAL source statements are numbered beginning with the START statement, given the number 0000, and incremented by one for each successive JOVIAL source statement encountered. To locate a statement in error in a program, the user counts down in the program listing the number of JOVIAL source statements specified in the error diagnostic, beginning with the START statement as statement zero. For example, given a user's time-sharing JOVIAL program containing the following:

```
00010   PROGRM TEST
00020    START
00030     TABLE TABA V 5 P $
00040       BEGIN ITEM TAA A 10 S 5 $
00050         BEGIN 1.0A5 2.0A5 3.00A5 4.0A5 END
00060             ITEM TAB A 20 S 5 $
00070        BEGIN 1084.80A5 1084.89A5 END
00080             ITEM TAC A 10 S 5 $
00090         BEGIN -8.0A5 END
00100     DEFINE IFEITHER "IFEITH" $
   .                 .
   .                 .
   .                 .
```

When the RUN command is given, the following error diagnostic is generated:

```
ERROR..STA..0005   MISSING END
```

There is no matched END for the BEGIN in line number 00040; the required END should be inserted as statement 0005 (line number 00100).

The correlation between line numbers and statement (STA) numbers of this sample program are as follows:

Line Numbers	Statement Numbers
00020	0000
00030	0001
00040	0002
00050	0002
00060	0003
00070	0003
00080	0004
00090	0004
00100	0005

TIME-SHARING JOVIAL SOURCE PROGRAM RESTRICTIONS

Source programs for time-sharing JOVIAL must adhere to the following restrictions:

1. Line lengths (including line number) cannot exceed 72 characters.
2. No COMDK source input is allowed.
3. No alter cards to update COMDKs are allowed.
4. No COMPOOL generation (GENCOM) is allowed.
5. No dynamic patching for the compiler is permitted.
6. No STC (Standard Transmission Code) encoding is allowed.

APPENDIX G

SYSTEM EDITOR INTERFACE

Version 49 of the JOVIAL compiler interfaces and interacts with the System Editor in a manner that allows the user to do source and object level edits and has the ability to create a variety of libraries and system files. Users are directed to the System Library Editor manual for detailed explanation of the various editing capabilities.

APPENDIX H

JOVIAL COMPILE ABORT CODES

The following are JOVIAL compiler abort codes and their explanations.

Code	Explanation
V0	Compiler error abort. Fatal error is encountered when processing direct code.
V1	Compiler error abort. An alternate construction of the indicated statement will probably allow it to compile.
V2	Object program abort. The JOVIAL compiler generates a MME GEBORT when replacing an erroneous JOVIAL statement.
V3	Compiler memory exhausted abort. The compiler needs more storage and its requests for additional memory have been denied. Therefore, additional space is not available to continue compilation.
V4	Compile activity has completed and preparation for the loader activity (binding the object H*) has aborted because not enough core is available to satisfy the CORE specification on the RUN command. The H* file will not be bound but, if on the RUN command a request was made to save a C* file, the C file for the object JOVIAL program will be created. The object program will not be placed in correction.
V5	Compiler error abort. Fatal error; an illegal construct has been encountered.

APPENDIX I

CROSS-REFERENCE TABLE

The JOVIAL cross-reference table is an alphabetized listing, by symbol, of a JOVIAL program's simple items, tables, table items (TITEM), arrays, files, statement labels, procedures (PROC), functions (FUNC), closes, and switch names. Each reference includes the following:

Symbol — The first 12 characters of the referenced name are included. Those constructs with more than 12 characters are shown truncated.

Type — If the construct is an array, item, table, table item, file or procedure, the programmers are so informed. Statement labels, switch labels, constructs defined within procedures, and external procedures are not identified under type.

Address — Addresses relative to the generated code are supplied for items, tables, arrays, files, and locally defined procedures. Externally defined procedures have their symref numbers in the address field. Table items, constructs defined within a procedure, and statement labels currently do not contain addresses.

Items extracted from a compool segment have the prefix "CP" on its address field. The address, if present, reflects its location relative to the origin of the labeled common area in which it is contained.

References — The references are JOVIAL source statement numbers. Statement numbers contained by asterisks (e.g., *12*) indicate that the referenced construct was declared within that statement. Numbers contained by dashes, (e.g., -12-) indicate that the value of the construct was set or re-set. All other numbers indicate that the construct was used in some fashion, but that its value was unchanged. Procedures that have no declaration reference are identified as external procedures and their address field contains a symref number.

The JOVIAL cross-reference table is automatically generated for any JOVIAL source program whenever the SYMTAB option is invoked on the $ JOVIAL control card.

INDEX

$
$ ENDJOB 11-5
$ EXECUTE 11-4
$ IDENT 11-1
$ JOVIAL 11-3
$ LIMITS 11-4
$ OPTION 11-2
$ SNUMB 11-1

'PROGRAM
'PROGRAM DECLARATION 6-6

***EOF
***EOF 11-6

ACCESSING
ACCESSING THE TIME-SHARING JOVIAL SUBSYSTEM F-10

ARITHMETIC
Arithmetic Operators 2-1

ARRAY
ARRAY DECLARATION 6-12

ASSIGN
Assign Statement 7-3
assign statement combinations 4-2

ASSIGNMENT
ASSIGNMENT STATEMENT 4-1

BATCH
Batch I/O F-7
Compile - Batch F-7

BODY
procedure body 7-8

BOOLEAN
Boolean 2-5
Boolean values 2-5

CLOSE
CLOSE DECLARATION 7-5

CODE
DIRECT CODE 7-1
DIRECT CODE RESTRICTIONS AND CONVENTIONS 7-1
Object Code Listing 1-3
Source Code Listing 1-3

CODING
input/output coding 9-5

COMMUNICATIONS
COMMUNICATIONS POOL 8-1

COMPILE
Compile - Batch F-7
Compile - Time-sharing F-7

COMPOUND
COMPOUND STATEMENT 4-6

CONSTANTS
CONSTANTS 2-5

DATA
PRESET DATA LISTS 6-5

DEBUG
DEBUG OPTION 10-1

DECLARATION
'PROGRAM DECLARATION 6-6
ARRAY DECLARATION 6-12
CLOSE DECLARATION 7-5
DEFINE DECLARATION 6-5
declaration list 7-8
FUNCTION DECLARATION 7-6
LIKE Table Declaration 6-11
MODE DECLARATION 6-4
Ordinary Table Declaration 6-7
OVERLAY DECLARATION 6-14
PROCEDURE DECLARATION 7-7
Specified Table Declaration 6-7
STRING Table Item Declaration 6-10
TABLE DECLARATION 6-6
Symbols Relating To Table Declarations 6-7

DEFINE
DEFINE DECLARATION 6-5

DEVICE
Device Manipulation 9-3

DIAGNOSTICS
INTERPRETING ERROR DIAGNOSTICS F-13

DIRECT
DIRECT CODE 7-1
DIRECT CODE RESTRICTIONS AND CONVENTIONS 7-1

DUMP
dump formats 7-11

ENCLOSURES
Enclosures 2-3

ENDJOB
$ ENDJOB 11-5

ERROR
INTERPRETING ERROR DIAGNOSTICS F-13

EXCHANGE
EXCHANGE STATEMENT 4-4

EXECUTE
$ EXECUTE 11-4

EXTERNAL
external subroutines 7-10

FILE
FILE CHARACTERISTICS F-7
FILE OPERATIONS 9-1
FILE STATEMENT 9-1

FIXED
Fixed point item 6-1
Fixed values 2-5

FLOATING
Floating point item 6-2
Floating values 2-5

FORMATS
dump formats 7-11

FUNCTION
FUNCTION DECLARATION 7-6
FUNCTIONS 3-4

GOTO
GOTO STATEMENT 4-4

I/O
Batch I/O F-7
PAPER TAPE I/O IN TIME-SHARING JOVIAL F-12
TERMINAL I/O F-10
Time-sharing I/O F-8

IDENT
$ IDENT 11-1

IFEITH
IFEITH STATEMENT 5-1

INPUT-OUTPUT
input-output parameters 7-7

INPUT/OUTPUT
input/output coding 9-5

INTEGER
Integer item 6-2
Integer values 2-5

ION
IOn STATEMENT 9-2

ITEM
Fixed point item 6-1
Floating point item 6-2
Integer item 6-2
ITEM DESCRIPTIONS 6-1
ITEM switch 4-4
Simple Item 6-1
STRING Table Item Declaration 6-10
Literal items 6-3
Status items 6-4

LANGUAGE
SOURCE LANGUAGE FORMAT 1-2

LIKE
LIKE Table Declaration 6-11

LIMITS
$ LIMITS 11-4

LIST
declaration list 7-8
Object Code Listing 1-3
Source Code Listing 1-3

LISTINGS
PROGRAM LISTINGS 1-3

LISTS
PRESET DATA LISTS 6-5

LITERAL
Literal 2-5
Literal items 6-3
Literal values 2-5

LOGICAL
Logical Operators 2-2

MODE
MODE DECLARATION 6-4

NAME
procedure name 7-7
STATEMENT NAME 4-1

NUMERIC
Numeric 2-5
Numeric values 2-5

OBJECT
Object Code Listing 1-3

OCTAL
Octal 2-6
Octal values 2-6

ONE-FACTOR
One-factor FOR statements 5-2

OPERATIONS
FILE OPERATIONS 9-1

OPERATORS
Arithmetic Operators 2-1
Logical Operators 2-2
Relational Operators 2-2

ORDINARY
Ordinary Table Declaration 6-7

OUT
OUT STATEMENT 9-5

OVERLAY
OVERLAY DECLARATION 6-14
Tables and Subordinate Overlays 6-9

PAPER
PAPER TAPE I/O IN TIME-SHARING JOVIAL F-12

PARALLEL
PARALLEL table 6-7

PARAMETERS
input-output parameters 7-7

POINT
Fixed point item 6-1
Floating point item 6-2

POOL
COMMUNICATIONS POOL 8-1

PRESET
PRESET DATA LISTS 6-5

PROCEDURE
PROCEDURE DECLARATION 7-7
procedure body 7-8
procedure name 7-7

PROGRAM
PROGRAM LISTINGS 1-3
SOURCE PROGRAM FORMAT 1-2
TIME-SHARING JOVIAL SOURCE PROGRAM RESTRICTIONS F-15

RELATIONAL
Relational Operators 2-2

RESERVED
Reserved JOVIAL Words 2-4

RUN
RUN COMMAND F-2
RUN EXAMPLES F-8

SEPARATORS
Separators 2-3

SERIAL
SERIAL table 6-6

SIGNS
SIGNS 2-1

SIMPLE
Simple Item 6-1

SNUMB
$ SNUMB 11-1

SOURCE
SOURCE LANGUAGE FORMAT 1-2
SOURCE PROGRAM FORMAT 1-2
Source Code Listing 1-3
TIME-SHARING JOVIAL SOURCE PROGRAM RESTRICTIONS F-15

SPECIAL
SPECIAL VARIABLES 3-2

SPECIFIED
Specified Table Declaration 6-7

STATEMENT
ASSIGNMENT STATEMENT 4-1
Assign Statement 7-3
assign statement combinations 4-2
COMPOUND STATEMENT 4-6
EXCHANGE STATEMENT 4-4
FILE STATEMENT 9-1
FOR STATEMENT 5-2
GOTO STATEMENT 4-4
IF STATEMENT 5-1
IFEITH STATEMENT 5-1
IN STATEMENT 9-4
IOn STATEMENT 9-2
OUT STATEMENT 9-5
STATEMENT NAME 4-1
STOP STATEMENT 4-5
SWITCH STATEMENT 4-4
TEST STATEMENT 5-4
WAIT STATEMENT 9-6
One-factor FOR statements 5-2
Three-factor FOR statements 5-3
Two-factor FOR statements 5-2

STATUS
Status 2-6
Status items 6-4
Status values 2-6

STOP
STOP STATEMENT 4-5

STRING
STRING Table Item Declaration 6-10

SUBORDINATE
Tables and Subordinate Overlays 6-9

SUBROUTINES
external subroutines 7-10

SWITCH
ITEM switch 4-4
SWITCH STATEMENT 4-4

SYMBOLS
Symbols Relating To Table Declarations 6-7

TABLE
LIKE Table Declaration 6-11
Ordinary Table Declaration 6-7
PARALLEL table 6-7
SERIAL table 6-6
Specified Table Declaration 6-7
STRING Table Item Declaration 6-10
Symbols Relating To Table Declarations 6-7
TABLE DECLARATION 6-6
Tables and Subordinate Overlays 6-9

TAPE
PAPER TAPE I/O IN TIME-SHARING JOVIAL F-12

TERMINAL
TERMINAL I/O F-10

TEST
TEST STATEMENT 5-4

THREE-FACTOR
Three-factor FOR statements 5-3

TIME-SHARING
Compile - Time-sharing F-7
Time-sharing I/O F-8

TIME-SHARING JOVIAL
ACCESSING THE TIME-SHARING JOVIAL SUBSYSTEM F-10
PAPER TAPE I/O IN TIME-SHARING JOVIAL F-12
TIME-SHARING JOVIAL SOURCE PROGRAM RESTRICTIONS F-15

TRANSMISSION
Transmission 9-3

TWO-FACTOR
Two-factor FOR statements 5-2

TYPE
Type Descriptor 2-4

UNDECLARED
undeclared variable 6-4

VALUES
Boolean values 2-5
Fixed values 2-5
Floating values 2-5
Integer values 2-5
Literal values 2-5
Numeric values 2-5
Octal values 2-6
Status values 2-6

VARIABLE
undeclared variable 6-4
SPECIAL VARIABLES 3-2
VARIABLES 3-1

WAIT
WAIT STATEMENT 9-6

WORDS
Reserved JOVIAL Words 2-4

Honeywell Bull

Repro-Service 8-73

HONEYWELL INFORMATION SYSTEMS

BS 06, Rev. 1